CAP. 2 Principii de bază în programarea orientată pe obiecte

În capitolul anterior am relevat faptul că principalele caracteristici ale *orientării obiectuale* gravitează în jurul conceptelor de tipuri *abstracte*, dinamica schimbului de *mesaje* şi extinderea funcționalității tipurilor existente prin derivarea lor în cadrul unor *ierarhii de moştenire*.

Am putea spune, fără să greşim, că programarea orientată pe obiecte, față de programarea structurată, formalizează în special latura descriptivă (statică) asigurând astfel o coerență sporită laturii procedurale (dinamice).

În continuare, în prima fază, ne vom ocupa de rezultatul direct al abstractizării care se remarcă prin definiția structurilor statice. Acestea sunt implementate destul de eterogen în limbajele declarate *OO,* unele dintre ele mixând în fapt modul de lucru tradițional cu structurile bazate pe obiecte, în timp ce altele optează ferm împotriva unui astfel de mod de abordare, considerându-se mai „pure" din perspectiva principiilor orientării obiectuale. Formalismul prezentat în continuare, pentru exemplificare, se sprijină în special pe constructorii limbajului *Java*.

Limbajul Java este un limbaj orientat obiect construit în primul rând să producă programe *sigure* adică să îngusteze cât mai mult căile de propagare a unor erori (inclusiv de proiectare) din faza de programare (elaborare a codului sursă) în faza de execuție. Compilatorul Java constituie în această privință elementul esențial, bazându-se pe o abordare puternic tipizată.

2.1 Clase şi metode. Instanțiere şi inițializare

În privința sctructurilor statice formalizate în urma unui proces de abstractizare, principiul de bază îl constituie principiul *încapsulării*.

2.1.1 Încapsulare

În POO datele (atributele sau variabilele membri) şi metodele (comportamentul) sunt integrate în *obiecte*; astfel datele şi metodele sunt legate intim împreună. Obiectele au proprietatea de a *ascunde informația* (information hiding), adică, deşi obiectele ştiu să comunice între ele prin intermediul *interfețelor*, în mod normal acestea nu conosc modul de implementare al serviciilor eferite de alte obiecte, detaliile de implementare fiind ascunse.

După cum am arătat în primul capitol, modelarea pe calea baza obiectelor se întemeiază pe posibilitatea definirii *tipurilor abstracte de date*. Stilul de programare care foloseşte abstractizarea datelor reprezintă o abordare metodologică care impune ca informația să fie *„ascunsă"* conştiincios într-un fragment de program. Mai exact, programatorul dezvoltă o serie de tipuri abstracte de date, fiecare putând fi privit din două perspective. Din exterior, un client (utilizator) al unui tip abstract de date poate „vedea" numai o colecție de operații care definesc comportamentul obiectului abstractizat. De cealaltă parte a interfeței, programatorul care a definit respectiva abstracțiune are în vedere variabilele utilizate pentru a menține starea internă a obiectului.

În concluzie, prin intermediul unui tip abstract de date este definită interfaţa unei abstracţiuni ale cărui detalii interne sunt *încapsulate,* cunoscute fiind doar programatorului care a definit respectiva abstracţiune. Se mai spune că *tipul* folosit de către un client (utilizator) îşi găseşte implementarea într-o *clasă* definită intern de către programator.

Pentru a desemna reprezentantul (sau un anumit exemplar) dintr-o clasă se foloseşte de obicei termenul de *instanţă* sau *obiect* concret. Obiectul este manipulat de obicei prin intermediul unei variabile de instanţă (*instance variable*) declarate ca fiind de tipul corespunzător clasei obiectului. De fapt, aceste variabile găzduiesc referinţe către obiectele instanţiate din clase.

Fiecare obiect (sau instanţă) are propriul set de *variabile de instanţă* ale căror valori îi reflectă starea sa internă. Acestea mai apar şi sub numele de *câmpuri* (legate mai mult de activitatea de modelare) sau *membri-date* (data fields/data members), în opoziţie cu *membrii-metode* care reprezintă operaţiile.

O viziune simplă dar completă a unei obiect presupune o combinaţie de *stare* şi *comportament*. Starea este descrisă prin intermediul câmpurilor sau variabilelor de instanţă, în timp ce comportamentul este caracterizat prin *metode*.

Deşi în programarea orientată obiect sunt în fapt create noi tipuri abstracte de date, în majoritatea limbajelor de programare este folosit termenul *clasă* pentru definirea acestora, şi termenul *tip* în declararea variabilelor prin care vor fi manipulate instanţele claselor. (există la un anumit nivel o distincţie şi mai subtilă între clasă şi tip, în special din perspectiva interfeţelor – un tip ar putea avea mai multe variante – clase – de implementare).

2.1.2 Definirea claselor

Clasele, la nivelul cărora sunt descrise toate detaliile „intime" ale tipurilor de obiecte, ar putea fi considerate asemeni unor tipare care determină trăsăturile definitorii ale instanţelor. Definiţia claselor va trebui să includă prin urmare toţi *membrii – variabile de instanţă* ale căror valori personalizează fiecare obiect, şi toţi *membrii – metode* care determină comportamentul obiectelor. Prin definiţia structurii claselor se vor delimita şi detaliile care formează „masca" vizibilă a fiecărui obiect de un anumit tip, facându-se astfel diferenţa între interfaţa (publică) şi mecanismele de implementare ale serviciilor specificate prin aceasta.

În limbaje de programare cum ar fi *Java* sau *C++* definirea claselor presupune declararea într-o manieră *descriptivă* a membrilor acestora într-un fragment de cod încadrat între marcatorii „{ }" (bloc unitar) şi care este desemnat prin numele clasei, prefixat cel puţin de cuvântul cheie *class*. *Delimitatorii „{ }"* mai sunt utilizaţi şi pentru a încadra atât instrucţiunile care formează corpul metodelor cât şi în cazul structurilor de controla pentru a determina blocul de instrucţiuni ce intră sun incidenţa acestora.

În figura de mai jos este prezentată definiţia clasei *Companie* din care se pot desprinde următoarea structură:

- membrii variabile de instanţă sunt *codFiscal, nume, sediu, departamente* şi *frmJrd*;

- membrii metode sunt *Companie* (constructorul clasei), *calculFondSalarii* şi *formaJuridică*.

```
public class Companie { // -> început definitie
  public String codFiscal;
  public String nume;
  public String sediu;
  public Departament[] departamente;
  private int formaJuridica;

  public Companie(String pCodFiscal, String pNume, String pSediu){}
  public double calculFondSalarii(){
    return fondSalarii;
  }
  public String getFormaJuridica(){
    if (formaJuridica == SA) return "Societate pe actiuni";
    if (formaJuridica == SA) return "Societate cu raspundere limitata";
     else return null;
   }

  public static final int SA = 1; //"Societate pe actiuni"
  public static final int SRL = 2; //"Societate cu raspundere limitata"
} // -> sfârşit definiţie
```

Figura 2-1 Definiţia clasei *Companie* în Java

Sintaxa folosită explicit pentru definirea unei variabile de instanţă se remarcă prin plasarea numelui variabilei la sfârşit, după lista de specificatori şi declaraţia *tipului*. Această declaraţie poate fi însoţită, bineînţeles, de o expresie de iniţializare introdusă prin operatorul de atribuire "=". De asemenea, mai trebuie specificat şi faptul că, în Java, încheierea unei instrucţiuni executabile se face prin caracterul ";".

Numele unei clase ar putea fi însoţit (prefixat), de asemenea, cu *specificatorul de vizibilitate* ***public***. În acest caz semnificaţia are legătură cu unitatea de organizare tipică în Java pentru grupurile de clase ce colaborează în acelaşi context, şi anume *pachetul* (package). Acest specificator are drept consecinţă şi definirea clasei pe care o însoţeşte în propriul fişier sursă *.java*, care reprezintă unitatea de compilare. *Package*-urile semnifică în Java unitatea de organizare modulară a claselor, circumscriindu-se de fapt unui *spaţiu de nume*. Clasele declarate publice pot fi aduse în spaţiul de nume al altei clase, externe package-ului în care au fost definite, *importând* spaţiul package-ului respectiv în întregime (*) sau parţial (doar clasele de interes).

După cum, de altfel, se observă şi în exemplul din figura 2-1, şi membrii unei clase (variabile sau metode) pot fi însoţiţi de ***specificatori de vizibilitate*** cum sunt:

- *public* – ce desemnează membrii vizibili de către toate clasele care au acces la respectiva clasă;
- *private* – ce desemnează membrii vizibili numai în interiorul clasei în care sunt definiți. Ei pot fi invocați numai de metodele (procedurile) interne.
- *protected* – pentru membrii care vor fi vizibili numai în clasele derivate din clasa respectivă. În Java specificatorul *protected* are sens în special dacă subclasele sunt definite în exteriorul package-ului clasei de bază, iar acestea au acces, prin *import*, la contextul acestui package.

În Java mai există un "gen" de vizibilitate specific, desemnat prin *ne-precizarea* vreunui astfel de specificator. Membrii în acest fel vor fi vizibili de către orice clasă internă package-ului respectiv, indiferent dacă specificatorul clasei este sau nu *public*. (detalii spuplimentare într-un paragraf următor)

De asemenea, membrii unei clase pot fi împărțiți în alte două categorii funcție de **specificatorul *static*** Astfel, anumite variabile-membru ar putea fi declarate ca având o singură "versiune" valabilă la nivelul clasei, toate instanțele respectivei clase partajând aceeaşi valoare pentru câmpul respectiv. Acest lucru este precizat prin prefixarea numelui membrului cu declarația (cuvântul cheie) *static*. Prin urmare membrii non-statici (care nu sunt declarați în mod explict ca *static*-i) vor avea "versiuni" distincte pentru fiecare instanță.

În definiția unei clase mai pot apare şi declarații *final* pentru membrii-variabile ale căror valori nu vor mai putea fi modificate după inițializare, deci pentru **constante**. În acest sens, pentru exemplul anterior formele juridice sunt definite prin două constante `SA` şi `SRL`. Aceste constante sunt declarate, de asemenea, statice şi, prin urmare, sunt valabile (în aceeaşi măsură) pentru toate instanțele, lucru explicabil dacă ne gândim că specificatorul *final* preîntâmpină modificarea valorii inițiale din definiția clasei.

Pe lângă membrii-variabile, în definiția claselor apar definite şi ***metodele***. În cazul acestora, *tipul* desemnează de fapt tipul (obiectului) returnat, iar dacă metodele nu au ca scop returnarea unei valori (sau instanțe) de un anumit tip, atunci tipul returnat este înlocuit prin cuvântul cheie *void*. Blocul de instrucțiunii care formează corpul metodelor este încadrat tot prin marcatorii "{ }" şi presupune redactarea unei secvențe de instrucțiuni inclusiv structuri de control şi declarații de variabile locale, variabile care nu sunt însă membri ai clasei ci sunt vizibile numai în spațiul de nume local al metodei respective.

Cel mai adesea membrii-variabile nu sunt accesebile din exteriorul clasei în mod direct, ci aceştia sunt declarați *private*, pentru accesul lor definindu-se două tipuri de metode: metode de acces – accesori (accessors) şi de modificare – modificatori (modifiers), care sunt disponibile prin interfața publică a tipului şi care pot fi invocate prin adresarea unor mesaje obişnuite.

În exemplul din figura de mai sus, informații despre membrul (privat) `formaJuridica` pot fi obținute prin metoda (publică) `getFormaJuridica()`.

Crearea unei clase nu se face niciodată de la zero. Întotdeauna o nouă clasă (altfel spus, definiția unui nou tip abstract de date) se bazează pe o clasă existentă pe care o extinde şi pe care

o va moşteni. Chiar dacă în declaraţia clasei nu există cuvântul cheie *extends* (în Java), care permite specificarea numelui clasei moştenite, nu înseamnă că respective clasă nu are "nici o bază". În astfel de cazuri se subînţelege că respectiva clasă este derivată din clasa *Object* ce consituie rădăcina ierarhiei de moştenire pentru sistemul de clase al mediului Java.

```
/** Persoana.java */

public class Persoana extends Object{ // definitie clasa Persoana
// membri-variabile publici
  public String cnp;
  public String numePren;
  public String domiciliu;

  protected Persoana(){/*constructorul clasei persoana*/}
} // sfarsit definitie clasa Persoana

----------------------------------------------------------------------
/** Salariat.java */

public class Salariat extends Persoana{ // definitie clasa Salariat
                                        // derivata din Persoana
// membri-variabile publici
  public String marca;
  public double pct_Vechime;
  public double pct_Penalizare;

// membri-variabile private
  private double salTarifar;
  private double pct_Conducere;
  private double sporVechime;
  private double sporConducere;
  protected double salariu;

// membri-metode
  public double calculSalariu() {return salariu;}
  public void stabVechime(boolean modifica, double pVechime){}
  public void stabConducere(boolean modifica, double pConducere){}

  public Salariat(String pCnp, String pNumePren, String pMarca,
    double pSalTarifar, double pVechime, double pConducere)
 {/*constructor clasa Salariat*/}
}
```

Figura 2-2 Crearea noilor clasele se realizează pe baza celor existente

În figura 3-2 clasa *Persoana* este constituită baza definiţiei clasei *Salariat* care va moşteni astfel toţi membrii clasei de bază (aspectele detaliate legate de moştenire le vom discuta într-o secţiune următoare). Clasa *Persoana* la rândul ei se bazează pe clasa *Object* (mai exact *java.lang.Object*) din sistemul de clase suport ale mediului de dezvoltare. În cazul în care o clasă de bază este construită direct din *Object*, atunci declaraţia `extends Object` poate fi omisă (este subînţeleasă de către compilator).

2.1.3 Metode, argumente, valori returnate

În programarea structurată termenul funcție sau procedură (de regulă diferența se face prin faptul că funcția returnează întotdeauna o valoare) este elementul definitoriu pentru organizarea (structurarea) logicii de prelucrare. În limbajele de programare orientate obiect, cum este Java, acest principiu se regăseşte sub forma *metodelor* (uneori apărând şi termenul membri-metode în contrapondere cu membrii-variabile).

Metodele determină în Java mesajele pe care le poate recepționa (la care poate răspunde) un obiect. Elementele cele mai importante ale declarației unei metode sunt *numele, argumentele, tipul returnat* şi *corpul* (blocul de instrucțiuni care formează partea executabilă a unei metode).

```
TipReturnat NumeMetodă (/*lista de argumente*/){
     /*Corpul_procedurii[1]*/
}
```

Apelul unei metode aparținând unui obiect se face printr-o sintaxă în care numele metodei urmează numelui obiectului de care aparține:

```
NumeObiect.numeMetodă(arg1, arg2, arg3);
```

Acțiunea prin care este apelată una din metodele unui obiect se numeşte *trimiterea sau expedierea unui mesaj*.

Lista de argumente specifică ce informație trebuie trimisă obiectului în cauză odată cu transmiterea mesajului. Această informație în Java, ca şi în alte limbaje de programare, poate lua forma transmiterii unor obiecte (sau, mai exact, a referințelor la anumite obiecte). Din acest motiv argumentele din specificația listei asociate unei metode trebuie însoțite de tipul care poate fi unul primitiv sau poate fi numele unei clase existente.

Dacă declarația unei metode este însoțită de numele unui tip (şi nu de cuvântul cheie *void*) atunci în corpul metodei trebuie specificată instrucțiunea ***return*** care realizează două acțiuni:

- în primul rând, la momentul *runtime* execuția metodei respective va fi întreruptă imediat după acestă instrucțiune;
- în al doilea rând, ca determina „producerea” unei valori (sau obiect) de tipul specificat în declarația metodei, care va fi preluată în instrucțiunea care a determinat apelul respectivei metode.

Dacă tipul returnat este specificat ca ***void*** atunci instrucțiunea *return* este folosită doar pentru a marca punctual de ieşire din program (sau, pur şi simplu, poate fi omisă).

Una din metodele speciale definite în interiorul unei clase şi care nu prezintă vreun tip returnat (nu este prefixată nici cu *void*), dar care a cărei nume este identic cu cel al clasei, este ***constructorul***. Rolul acestei metode este să inițializeze obiectele create pe baza respectivei clase.

Prin urmare această metodă va fi invocată de către o instrucţiune de instanţiere care, în Java, ia forma următoare:

```
NumeClasă nume_variabilă = new NumeClasă();
```

în care numele tipului variabilei, ce va găzdui referinţa către obiectul creat, este numele clasei şi care va corespunde cu numele constructorului introdus prin cuvântului cheie ***new***. Constructorii, ca oricare alte metode, pot fi parametrizate, astfel că valorile transmise în mometul instanţierii să fie folosite pentru a determina starea iniţială a obiectelor. Iată spre exemplu crearea unui salariat (vezi definiţia din figura 3-2):

```
Salariat sal = new Salariat("CNPXXX", "Ion Ion", "1001",
3500000, .15, 0);
```

Dacă în interiorul clasei nu este definit explicit un constructor, iniţializarea va fi garantată printr-un constructor implicit furnizat de mediul Java.

La fel ca şi în cazul membrilor-variabile sau câmpuri, o metodă poate fi declarată ca ***static***-ă. În principal această declaraţie s-ar putea traduce prin faptul că respectiva metodă nu este legată de un anumit obiect concret ci este definită la nivel de clasă. Implicaţia imediată este că respectiva metodă este apelabilă în lipsa vreunei instanţe din clasa respectivă[2], iar ca urmare toate metodele non-statice nu pot fi apelate decât prin intermediul instanţelor. O altă consecinţă este şi faptul că metodele non-*statice* (şi nici câmpurile non-*statice*) nu pot fi apelate în metode *statice*. Astfel la compilarea claselor următoare, apelul `metodaNonStatica();` din `metodaStatica()`:

```
public class TestStatic {
  static void metodaStatica(){
    metodaNonStatica();
    System.out.println("Static Success");
  }
  void metodaNonStatica(){
    System.out.println("NonStatic Success");
  }
  public static void main(String[] args) {
    TestStatic t = new TestStatic ();
    t.metodaNonStatica();
  }
}
```

Figura 2-3 Tentativa de invocare context non-static din context static

va produce următoarea eroare de compilare:

```
Error #: 308 : non-static method metodaNonStatica() cannot be
referenced from a static context
```

[1] În Java comentariile se realizează prefixând liniile respective cu şirul //, sau încadrarea lor într-un bloc delimitat prin /* şi */.

[2] În acest caz este oarecum o încălcare a paradigmei obiect-mesaj, această caracteristică nefiind considerată pur orientată-obiect. În schimb însă, chiar în limbajele de modelare (în particular în UML) atributele şi metodele pot fi definite cu scop la nivel de *clasificator* (clasă) sau la nivel de *instanţă* (vezi capitolele următoare).

2.1.3.1 Inițierea execuției în Java

Execuția unui program Java înseamnă practic încărcarea unei clase şi lansarea unei metode *statice* speciale datorită faptului că în mediul Java toate instrucțiunile sunt, într-un fel sau altul, incluse în definiția claselor. Cu alte cuvinte, nu există fragmente de cod executabil care să se regăsească în afara definiției unei clase. Şi totuşi cum se inițiază execuția în Java ? Mediul run-time al Java (JRE – Java Runtime Engine care se bazează în fapt pe o „maşină virtuală” ce asigură portabilitatea codului pe (aproape) orice platformă) acceptă numele unei clase pe care încearcă apoi să o localizeze şi să o încarce, după care inițierea execuției propriu-zise înseamnă căutarea (în definiția respectivei clase) şi lansarea metodei cu semnătura ***private static void main(String[ ] args)***. Inexistența metodei ***main*** sau existența unei metode cu acest nume dar fără semnătură corectă (de exemplu fără parametrul de tip *String []*[3]) duce la imposibilitatea localizării punctului de declanşare a execuției. Clasa *TestStatic* din figura 2-3 prezintă o astfel de metodă *main*, deci ar putea fi lansată în execuție (bineînțeles după ce compilatorul a validat-o ceea ce nu înseamnă că la execuție n-ar mai putea apare anumite erori).

2.1.4 Instanțierea claselor şi supraîncărcarea constructorilor

Constructorul reprezintă o metodă asociată oricărei clase şi care este apelată în momentul creării instanțelor. Dacă programatorul nu furnizează o definiție explicită pentru un constructor, atunci obiectele clasei respective vor fi create folosind *constructorul implicit* (constructorul *default* fără argumente).

Numele constructorului desemnează o funcție-membru cu numele clasei, şi nu prezintă nici un tip returnat (nici măcar **void**).

2.1.4.1 Definirea constructorilor

La fel ca în cazul oricăror altor metode, constructorul poate avea argumente care să permită specificarea modului în care vor fi create obiectele, cu alte cuvinte constructorul desemnează o cale prin care se poate parametriza inițializarea obiectelor.

[3] vector cu elemente de tip *String*. Clasa *String* se regăseşte printre clasele suport ale limbajului Java

```
public class Companie {
  public String codFiscal;
  public String nume;
  public String sediu;
  public Departament[] departamente;
... ... ... ...
// declarația completă cu parametri a constructorului
  public Companie(String pCodFiscal, String pNume, String pSediu){
// setul de instrucțiuni prin care sunt inițializați membrii
    codFiscal = pCodFiscal;
    nume = pNume;
    sediu = pSediu;
  }
}
```

Figura 2-4 Clasa *Companie* cu definiția completă a unui constructor parametrizat

Astfel pe baza definiției din figura 3.4 următoarea secvența de cod:

```
public class Test {
  public static void main(String[] args) {
    Companie firma = new Companie("R00001", "Firma mea SRL", "Iasi, Copou 33");
    System.out.println("Am creat " + firma.nume + " din " + firma.sediu);
  }
}
```

va produce rezultatul:

```
Am creat Firma mea SRL din Iasi, Copou 33
```

Specificarea explicită în cadrul definiței clasei a unui constructor (parametrizat sau nu) suprascrie constructorul implicit furnizat de mediul Java. Cu alte cuvinte existența unor constructori definiți în cadrul clasei constrânge programatorul să folosească unul dintre aceştia la instanțierea obiectelor. Altfel, următoarea secvență de cod:

```
public class Test {
  public static void main(String[] args) {
    Companie firma = new Companie();
  }
}
```

va produce eroarea:

```
"Test.java": Error #: 300 : constructor Companie() not found
in class prosal.salariati.Companie
```

2.1.4.2 Supraîncărcarea constructorilor

În Java, ***supraîncărcarea metodelor*** se referă, în general, la posibilitatea folosirii aceluiaşi nume pentru mai multe metode diferite în cadrul aceleaşi clase. Un caz de supraîncărcare des întâlnit se referă la *supraîncărcarea constructorilor*. Astfel, pe de o parte găsim constructorul *default* – constructorul fără argumente, şi pe de altă parte constructorii parametrizați definiți explicit. Pot exista mai mulți astfel de constructori parametrizați, definiți pentru aceeaşi clasă şi diferențiați prin tipul sau poziția argumentelor, toți având acelaşi nume, adică cel al clasei.

```
class Departament {
  public String nume;
  public int nrMembri;

  public Departament(String pNume, int pNrMembri){
    nume = pNume;
    nrMembri = pNrMembri;
  }
}

public class Companie {
  public String codFiscal;
  public String nume;
  public String sediu;
  public Departament[] departamente;

// constructorul default
  public Companie(){
    System.out.println("Instantiere Companie prin constructor fara
                      parametri");
    System.out.println(nume);
  }
// primul constructor parametrizat
  public Companie(String pCodFiscal, String pNume, String pSediu){
    System.out.println("Instantiere Companie - primul constructor cu
                      parametri");
    codFiscal = pCodFiscal;
    nume = pNume;
    sediu = pSediu;
    System.out.println(nume);
  }
// al doilea constructor parametrizat
  public Companie(String pCodFiscal, String pNume,
                 String pSediu, int nrDepartamente){
    System.out.println("Instantiere Companie - al doilea constructor cu
                      parametri");
    codFiscal = pCodFiscal;
    nume = pNume;
    sediu = pSediu;
    departamente = new Departament[nrDepartamente];
    System.out.println(nume);
  }

  public static void main(String[] args){
    Companie prima_firma = new Companie();
    Companie a_doua_firma = new Companie("R0001","A doua Firma", null);
    Companie a_treia_firma = new Companie("R0001","A treia Firma",
                                        null, 2);
  }
}
```

Figura 2-5 Supra încărcarea constructorilor

Iar rezultatul va fi:

```
Instantiere Companie prin constructor fara parametri
null
Instantiere Companie - primul constructor cu parametri
A doua Firma
Instantiere Companie - al doilea constructor cu parametri
A treia Firma
```

Diferențierea *metodelor supraîncărcate* este posibilă prin impunerea restricției ca fiecare să aibă o ***listă de argumente unică*** prin: (1) tipul argumentelor, (2) ordinea argumentelor (nu se iau în considerare numele argumentelor şi nici tipul returnat). De asemenea, nu este posibilă supraîncărcarea prin tipul returnat.

Dacă se crează o clasă fără nici un constructor, atunci compilatorul va crea în mod automat un constructor *default*.

Atenție: invocarea unui constructor cu o listă de argumente care nu se potriveşte nici unui constructor definit (sau default*) va genera o eroare de compilare.*

2.1.4.3 Auroreferențierea

Cuvântul cheie **this** (care semnifică "acest obiect" sau "obiectul curent") poate fi utilizat numai în cadrul unei metode non-statice şi produce referința către obiectul a cărui metodă a fost apelată. Apelul unei anumite metode în cadul unei alte metode aparținând aceluiaşi obiect nu implică folosirea obligatorie a cuvântului cheie *this*. Acest cuvânt este folosit în special pentru instrucțiunea *return* care trebuie să returneze referința obiectului curent. Altfel spus *this* returnează referința obiectului a cărui metodă a fost apelată.

This poate fi apelat din interiorul unui constructor, caz în care este folosit sub forma unui apel cu argumente – ***this (arg)*** – şi va invoca în mod explicit constructorul corespunzător listei de argumente.

Există posibilitatea ca în interiorul unui constructor să fie apelat un alt constructor, caz în care cuvântul cheie *this* este folosit pentru a face apel către ceilalți constructori. Spre exemplu programul din figura 2-5 poate fi simplificat astfel:

```
class Departament {
  public String nume;
  public int nrMembri;

  public Departament(String pNume, int pNrMembri){
    nume = pNume;
    nrMembri = pNrMembri;
  }
}

public class Companie {
  public String codFiscal;
  public String nume;
  public String sediu;
  public Departament[] departamente;

// constructorul default
  public Companie(){
    System.out.println("Instantiere Companie prin constructor fara
                        parametri");
    System.out.println(nume);
  }
// primul constructor parametrizat
  public Companie(String pCodFiscal, String pNume, String pSediu){
    System.out.println("Instantiere Companie - primul constructor cu
                        parametri");
    codFiscal = pCodFiscal;
    nume = pNume;
    sediu = pSediu;
    System.out.println(nume);
  }
// al doilea constructor parametrizat
  public Companie(String pCodFiscal, String pNume,
                  String pSediu, int nrDepartamente){
    this(pCodFiscal, pNume, pSediu);
    System.out.println("Instantiere Companie - al doilea constructor cu
                        parametri");
    departamente = new Departament[nrDepartamente];
    System.out.println(nume);
  }

  public static void main(String[] args){
    Companie prima_firma = new Companie();
    Companie a_doua_firma = new Companie("R0001","A doua Firma", null);
    Companie a_treia_firma = new Companie("R0001","A treia Firma",
                                          null, 2);
  }
}
```

Figura 2-6 Definiția clasei *Companie* modificată pentru al doilea constructor parametrizat

iar rezultatul va fi:

```
Instantiere Companie prin constructor fara parametri
null
Instantiere Companie - primul constructor cu parametri
A doua Firma
Instantiere Companie - primul constructor cu parametri
A treia Firma
Instantiere Companie - al doilea constructor cu parametri
A treia Firma
```

Din rezultat se poate observa dubla apelare a primului constructor parametrizat ca urmare a apelului acestuia şi de către al doilea constructor pentru creare celei de a treia firme.

Cuvântului cheie **static** se referă la metodele (membrii) care nu vor putea fi invocate prin ***this***. O metodă (sau membru) declarată *static* este creată la nivelul clasei şi nu la nivelul fiecărui obiect iniţializat – echivalentul unei funcţii globale. Ca urmare metoda statică sau membrul static este disponibil(ă) în momentul încărcării clasei indiferent dacă aceasta a fost instanţiată sau nu. Metodele statice pot face referire sau pot apela alte metode sau câmpuri *static-e*, dar nu şi non-statice. O metodă non-statică poate face însă referire la un membru static.

2.1.5 Inițializarea membrilor

Iniţializarea membrilor declaraţi în definiţia unei clase se face diferit funcţie de cei *statici* şi cei *non-statici*. Cei statici sunt iniţializaţi la încărcarea clasei în memorie, iar cei non-statici sunt iniţializaţi la instanţierea clasei – prin urmare sunt iniţializaţi pentru fiecare obiect nou creat.

Java *garantează* faptul că variabilele sunt într-un fel sau altul iniţializate: dacă este vorba despre o variabilă locală non-primitivă compilatorul obligă programatorul să o iniţializeze înainte de a o folosi (altfel se generează un mesaj de eroare de compilare). Dacă este vorba de un membru primitiv atunci, în situaţia în care nu sunt iniţializaţi explicit, Java îi iniţializează cu valori default (`0` pentru primitive numerice, `0` sau spaţiu pentru primitive `char`, `false` pentru primitive `boolean` şi `null` pentru referinţe).

Ordinea de iniţializare a variabilelor este determinată de ordinea în care acestea sunt definite în interiorul clasei. Definiţiile variabilelor pot fi răspândite în mai multe locaţii ale definiţiei clasei şi în metodele care-i aparţin, însă variabilele sunt întotdeauna iniţializate înainte de a fi invocate.

Iniţializarea membrilor referinţe presupune folosirea unei sintaxe care să implice invocarea unui constructor, cum ar fi:

```
ref = new ClassName(arguments);
```

Iniţializarea membrilor (non-statici) se poate face atât *explicit la declararea acestora în definiţia clasei* cât şi *în interiorul constructorului*. Ordinea de iniţializare presupune că iniţializarea automată la declararea membrilor precede întotdeauna iniţializarea din interiorul constructorului.

2.1.5.1 Inițializarea datelor (membrilor-câmpuri) statice

În cazul membrilor-variabile ***static***-e primitive neinițializate explicit, acestea vor primi valorile standard ca şi în cazul celor non-statice. Dacă însă este vorba despre membri statici declarați ca referințe pentru obiecte atunci, în cazul neinițializării lor (prin invocarea explicită a constructorului corespunzător), vor primi *null*.

Invocarea pentru prima dată a unei clase presupune încărcarea acesteia. Înainte de a rezolva orice apel, se inițializează membrii statici (care sunt unici, nu se multiplică funcție de numărul de obiecte instanțiate). Crearea unui nou obiect din clasa respectivă nu presupune reinițializarea membrilor statici, doar membrii non-statici sunt reinițilizați pentru fiecare instanță în parte. În acest sens edificator este exemplul care urmează.

```
class Pagina {
  static Titlu antet = CarteInfo.titluCarte;
  int nrPagina;
  static int nrPaginiCarte = CarteInfo.nrPagini;
  Pagina(int pageno){
    nrPagina = pageno;
    System.out.println("Pagina["+nrPagina+"].nrPaginiCarte
"+nrPaginiCarte);
  }
}
class Titlu {
  String autor;
  String titlu;
  Titlu(String pautor, String ptitlu){
    autor = pautor;
    titlu = ptitlu;
    System.out.println("Am initializat titlu "+autor+" "+titlu);
  }
}

public class CarteInfo {
  static Titlu titluCarte = new Titlu("Eckel", "Java");
  Pagina[] pagini;
  static int nrPagini;
  CarteInfo(int pnrpagini){
    nrPagini = pnrpagini;
    pagini = new Pagina[nrPagini];
    for(int i = 0; i <= pnrpagini - 1; i++)
      pagini[i] = new Pagina(i+1);
  }
  public static void main(String[] args) {
    System.out.println("1: CarteInfo.Pagini "+ nrPagini);
    CarteInfo o_carte = new CarteInfo(2);
    System.out.println("2: CarteInfo.nrPagini "+ nrPagini);
    }
}
```

Figura 2-7 Inițializarea membrilor statici

Rezultatul exemplului din figura 3-7 este următorul:

```
Am initializat titlu Eckel Java
1: CarteInfo.Pagini 0
```

```
Pagina[1].nrPaginiCarte 2
Pagina[2].nrPaginiCarte 2
2: CarteInfo.nrPagini 2
```

2.1.5.2 Inițializarea vectorilor sau tablourilor

În mod obişnuit orice programator a lucrat cu tablouri ale căror elemente însă erau doar tipuri primitive. În limbajele de programare orientate pe obiecte posibilitatea definirii tipurilor abstracte de date deschide calea creării tablourilor ale căror elemente sunt obiecte de tipul unor clase existente. Sintaxa generală de declarare a unui astfel de tablou este, în linii mari, următoarea:

```
NumeClasă [] numeTablou; // declarea unui array
```

Se observă că tablourile sunt însoțite, în momentul definirii dar şi al utilizării, de paranteze pătrate [] (nu rotunde) pentru indicarea operatorului de indexare.

Pentru aflarea dimensiunii (numărului de elemente) tablourile prezintă proprietate instrinsecă ***lenght***. Această proprietate nu poate fi schimbată în mod direct însă poate fi accesată. Indexarea elementelor unui tablou începe de la 0, prin urmare valoarea maximă a indexului este *lenght – 1*.

Elementele care formează un tablou pot fi primitive sau pot fi obiecte (mai exact referințe la obiecte existente).

Operația de inițializare a unui tablou poate fi sintetizată în două etape:

1. inițializarea tabloului "în sine" - în această etapă va fi specificat numărul de elemente care vor forma tabloul şi
2. (cu valori primitive sau referințe de obiecte)

```
// prima variantă: inițializarea separată a tabloului şi a elementelor
int [] a1 = new int[3];
a1 [1] = 1, a1[2] = 2, a1[3] = 3;
// a două variantă: inițializarea în aceeaşi instrucțiune a tabloului şi elementelor
int [] a1 = {1, 2, 3}
```

Inițializarea unui tablou se poate face şi prin folosirea cuvântului rezervat ***new*** atât pentru referințe cât şi pentru obiecte.

```
// pentru un tablou de primitive
int [] a1 = new int [3];
// pentru un tablou de obiecte
Integer [] a2 = new Integer [3];
// inițializarea explictă a elementelor
a2 [0] = new Integer(1);
a2 [1] = new Integer(2);
a2 [2] = new Integer(3);
```

Dacă inițializarea unui tablou nu este urmată sau nu implică şi inițializarea explicită a elementelor sale, acestea vor inițializate cu valorile implicite (în cazul tablourilor de obiecte se obțin în această primă fază referințe nule).

```
// O altă formă de comprimare a definirii şi inițializării unui tablou
```

```
Integer [] b = new Integer [] { new Integer(1), new Integer(2), new Integer(3)};
// sau
Integer [] c = { new Integer(1), new Integer(2), new Integer(3)};
```

Dacă tabloul nu a fost inițializat în prealabil, inițializarea separată a elementelor sale nu poate avea loc. Adică inițializarea unui tablou (precizarea tipului elementelor şi a dimensiunii - *length*) precede inițializarea elementelor sale individuale. De exemplu secvența următoare va genera o excepție în momentul compilării, reclamând inexistența variabilei `a2`:

```
// definim tabloul fără inițializare
Integer [] a2;
// inițializarea explictă a elementelor
a2 [0] = new Integer(1);
a2 [1] = new Integer(2);
a2 [2] = new Integer(3);
```

Eroarea rezultată este:

```
Error #: 553 : variable a2 might not have been initialized at
line 38, column 1
```

2.1.6 Vizibilitate – controlul accesului

În limbajele de programare orientate obiect, în speță Java, vizibilitatea comportă două aspecte: pe de o parte este vorba despre aplicarea principiului ascunderii informației la nivelul membrilor-variabile sau metode şi, pe de altă parte, este vorba despre formarea "spațiilor de nume" la nivelul organizării claselor în module (pachete sau package-uri în Java) funcționale.

2.1.6.1 Controlul accesului la membrii claselor

Specificatorii pentru acces `public`, `protected`, `private`, amintiți deja, au ca principal rol separarea elementelor stabile, care pot fi invocate de "clienții" bibliotecilor de clase, față de elementele care se găsesc sub controlul exclusiv al celor care se ocupă de implementare. Altfel spus, este vorba despre separarea între elementele disponibile – ale căror specificații sunt declarate utilizabile în contextul exterior – şi cele ascunse – care reprezintă în cea mai mare măsură detalii de implementare.

Pentru a acorda acces la un membru al unei clase, în Java există patru căi:

1. declararea acestuia ca ***public*** pentru a fi vizibil din orice context sau spațiu de nume din care face parte tipul respectivei clase. Declararea metodelor unei clase ca *publice* se face în scopul constituirii unei viziuni a serviciilor oferite de respectiva clasă, cu alte cuvinte se defineşte *interfața publică* a clasei.
2. definirea acestuia în mod *friendly*, adică fără să fie însoțit de vreo indicație explicită privind vizibilitatea, ceea ce îl face disponibil în contextul package-ului în care este declarată clasa din definiția căreia face parte.
3. declararea acestuia ca ***protected*** ceea ce-l îl va face vizibil numai pentru clasele derivate (care moştenesc) clasa de bază în care a fost declarat

4. furnizarea unor de metode de acces sau modificare care să fie disponibile (public, friendly, protected) în domeniul de vizibilitate desemnat, şi care vor defini şi controla modul în care trebuie să fie consultat şi modificat.

*Un membru declarat **public** este disponibil oricărei clase care importă package-ul în care este declarat.* (despre package-uri vezi în paragraful următor)

Clienții claselor nu au nevoie să cunoască detaliile prin care serviciile oferite prin interfața publică sunt realizate. De aceea membrii interni, la fel ca şi definițiile metodelor private, nu sunt accesibile clienților claselor. Dacă se doreşte ca un membru să fie vizibil numai intern, în cadrul specificațiilor de implementare, şi invizibil sau indisponibil în nici un fel în afară, se foloseşte declarația ***private***. În mod normal secvența următoare:

```
class Salariat {
  public String marca;
  public double pct_Vechime;
  Salariat (String pMarca, double pVechime){
    marca = pMarca;
    pct_Vechime = pVechime;
  }
}

public class Test {
  public static void main(String[] args) {
    Salariat sal = new Salariat("1001", .15);
    System.out.println("Salariatului cu marca " + sal.marca +
    " i se va aplica un procent spor vechime de " +  sal.pct_Vechime);
  }
}
```

va produce următoarul rezultat:

```
Salariatului cu marca 1001 i se va aplica un procent spor conducere de 0.15
```

Dacă însă se doreşte păstrarea confidențialității asupra sporului de conducere atunci membrul *pct_Conducere* poate fi declarat *private* redefinind astfel clasa *Salariat*:

```
class Salariat{
  public String marca;
  private double pct_Conducere;
  Salariat (String pMarca, double pConducere){
    marca = pMarca;
    pct_Conducere = pConducere;
// la acest nivel am acces la membrii privați
    System.out.println("Salariatului cu marca " + this.marca +
    " i se va aplica un procent spor conducere de " +  this.pct_Conducere);
  }
}
```

iar la compilarea clasei *Test* vom obține următoarea eroare ca urmare a faptului că în metoda *main* a acesteia se încearcă accesarea unui membru privat:

```
"Test.java": Error #: 306 : variable pct_Conducere has private access in class teste.Salariat
```

deşi inițializarea acestuia este posibilă prin intermediul constructorului *Salariat* care are acces intern la membrii clasei de care aparține. Mutând însă instrucțiunea de afişare din clasa *Test.main()* în constructorul *Salariat* (binenînțeles eliminând numele variabilei *sal* sau înlocuindu-l cu *this*) vom obține un rezultat pozitiv.

Prin urmare, declarația **private** semnifică faptul că membrul respectiv este inaccesibil în afara clasei în care este definit. Metodele declarate **private** sunt considerate utilitare, având rol doar în implementare şi nu sunt accesibile prin intermediul vreunei interfețe.

Controlul instanțierii claselor sau, altfel spus, prevenirea instanțierii directe, poate fi realizat(ă) prin declararea ca **private** a *constructorilor* şi apelarea acestora doar prin intermediul unor metode speciale vizibile prin declarații **public**. De exemplu:

```
class Conexiune {
  private Conexiune() {};
  static Conexiune Conecteaza(){
    return new Conexiune();
  }
}
public class AplicatieBD {
  public static void main(String[] args) {
    //Conexiune cnx = new Conexiune(); nu merge
    Conexiune cnx = Conexiune.Conecteaza();
  }
}
```

În cazul în care o clasă este derivată (moşteneşte) dintr-o clasă de bază definită într-un alt pachet (package), atunci aceasta nu are implicit acces decât la interfața publică a clasei de bază, ne-beneficiind şi de accesul „friendly” la membrii necalificați printr-o declarație de vizibilitate. Astfel deşi clasa derivată moşteşte toți membrii aceasta nu are acces la cei „friendly” chiar dacă aceştia sunt vizibili de către clasele din package-ul din care face parte clasa de bază. Pentru a extinde domeniul de vizibilitate al acestor membri şi în clasele derivate care nu se găsesc în acelaşi package cu clasa de bază, în Java se foloseşte specificatorul **protected**. Prin urmare specificatorul **protected** nu schimbă vizibilitatea acestor membri în package-ul clasei de bază ci doar extinde vizibilitatea lor şi pentru clasele derivate din afara lui.

Metode de acces şi de modificare

După cum am precizat mai sus variabilele *private* pot fi manipulate de către metodele clasei în care sunt definite. În practică se mai obişnuieşte ca membrii ale căror valori sunt stabilite de către clienți (extern) să fie declarați totuşi *private*, iar manipularea lor este lăsată în seama unor metode publice special dedicate acțiunilor de acces sau modificare a respectivilor membri. Aceste metode (*accesori – accesors/query* şi *modificatori-modifiers/mutators* sau *get* şi *set* – după cum sunt de obicei prefixate) au avantajul că înainte de modificarea efectivă a membrilor pot efectua diferite verificări conforme cu anumite restricții impuse de logica aplicațiilor.

Un exemplu edificator ar fi clasa *Time* care permite specificarea orei, minutului şi secundei, însă cu respectarea restricțiilor 0<=ora<=24, 0<=minutul<=60, 0<=secunda <=60:

```
public class Time {
    private int ora;      // 0 - 23
    private int minutul;   // 0 - 59
    private int secunda;   // 0 - 59

    // Constructorii
    public Time() { setTime( 0, 0, 0 ); }
    public Time( int h ) { setTime( h, 0, 0 ); }
    public Time( int h, int m ) { setTime( h, m, 0 ); }
    public Time( int h, int m, int s ) { setTime( h, m, s ); }
    public Time( Time timp )
    {
       setTime( timp.getOra(),
                timp.getMinutul(),
                timp.getSecunda() );
    }

    // Metodele Set
    // Stabileste noua valoare pentru timp. Efectueaza
    // verificari de validitate asupra datelor
    public void setTime( int h, int m, int s )
    {
       setOra( h );    // set the Ora
       setMinutul( m );  // set the Minutul
       setSecunda( s );  // set the Secunda
    }
    // set pentru ora
    public void setOra( int h )
       { ora = ( ( h >= 0 && h < 24 ) ? h : 0 ); }
    // set pentru minut
    public void setMinutul( int m )
       { minutul = ( ( m >= 0 && m < 60 ) ? m : 0 ); }
    // set pentru secunda
    public void setSecunda( int s )
       { secunda = ( ( s >= 0 && s < 60 ) ? s : 0 ); }

    // Metode get
    // get pentru ora
    public int getOra() { return ora; }
    // get pentru minut
    public int getMinutul() { return minutul; }
    // get pentru secunda
    public int getSecunda() { return secunda; }
 }
```

Figura 2-8 Clasa *Time* cu membri privați şi metode *set* şi *get* publice

Alt exemplu clasic, ceva mai apropiat de problemele economice, ar fi cel legat de debitarea sau creditarea unui cont bancar, în care manipularea disponibilului, debitului, creditului şi soldului constituie detalii interne manipulate prin operații publice care reflectă operațiile ce pot fi realizate de către un client:

```
public class Cont{
  public String nrCont;
  public String tipClient;
  public String titularCont;
  public String tipCont;

  private float debitCont;
  private float creditCont;
  private float limitaCreditare;
  private float disponibil;
  private float soldCont;

  public float depune(float sumaDepusa){
    debitCont = (creditCont > sumaDepusa) ? 0 : debitCont +
              (sumaDepusa - creditCont);
    creditCont = (creditCont > sumaDepusa) ? creditCont - sumaDepusa : 0;
    soldCont = debitCont - creditCont;
    disponibil = limitaCreditare + soldCont;
    System.out.println("Varsat in cont pentru " + titularCont + " suma de "
    + sumaDepusa + " Sold cont " + soldCont + " Disponibil " + disponibil);
    return soldCont;
  }
  public float retrage(float sumaRetrasa){
    if (sumaRetrasa > disponibil)
      System.out.println("Fonduri insuficiente. Poate fi retrasa o suma maxima de "
        + disponibil);
    else {
      debitCont = (sumaRetrasa >= debitCont) ? 0 : debitCont - sumaRetrasa;
      creditCont = (sumaRetrasa > debitCont) ? creditCont +
                   (sumaRetrasa - debitCont) : 0;
      disponibil = disponibil + debitCont - creditCont;
      soldCont = debitCont - creditCont;
      System.out.println("Retras din contul " + titularCont + " suma de "
        + sumaRetrasa + " Sold cont " + soldCont + " Disponibil " + disponibil);
      }
    return soldCont;
  }
  public void stabLimitaCreditare (float suma) {
    if (tipCont == "Credit") {
      limitaCreditare = suma;
      disponibil = limitaCreditare + soldCont;
      System.out.println("Am creditat contul " + titularCont + " cu suma de "
        + suma + " Sold cont " + soldCont + " Disponibil " + disponibil);
    }
  }
}
```

Figura 2-9 Clasa *Cont* cu membri privaţi şi operaţii de manipulare publice

Dacă vom lansa în execuţie fragmentul de cod:

```
public class TestCont {
  public static void main(String[] args) {
    Cont c = new Cont();
    c.nrCont = "1111112211";
    c.titularCont = "Ion Ion";
    c.tipCont = "Credit";
    c.stabLimitaCreditare(2000000);
    c.depune(1000000);
    c.retrage(2000000);
  }
}
```

rezultatul ar fi:

```
Am creditat contul Ion Ion cu suma de 2000000.0 Sold cont 0.0 Disponibil 2000000.0
Vărsat in cont pentru Ion Ion suma de 1000000.0 Sold cont 1000000.0 Disponibil 3000000.0
Retras din contul Ion Ion suma de 2000000.0 Sold cont -2000000.0 Disponibil 1000000.0
```

2.1.6.2 Pachete şi controlul accesului la clase

Şi în limbajele de programare orientate obiect există posibilitatea organizării funcţionalităţii în unităţi specifice care formează *spaţii de nume*. Elementele care formează baza acesteia organizării modulare vor fi însă clasele. Vizibilitatea inter-modulară ia, în acest context, forma vizibilităţii claselor din exteriorul modulelor din care fac parte.

În Java unitatea de organizare a bibliotecilor de clase este pachetul sau *package-ul*, iar accesul la clasele din interiorul spaţiului de nume desemnat printr-un pachet se face prin *import*-ul claselor care au asociat specificatorul de vizibilitate *public*.

Package-ul – unitatea de organizare a bibliotecilor de componente în Java

Un *spaţiu de nume* presupune că *fiecare element are un nume unic în interiorul acestuia*, iar în exterior numele elementelor vor putea fi invocate (dacă, bineînţeles, sunt publice) specificând (mai ales pentru a evita conflictele de nume) *numele intern calificat prin prefixare cu numele spaţiului de nume*.

Prin urmare, în cazul Java numele claselor *trebuie să fie unice* în cadrul package-urilor în care sunt definite, iar în exterior vor fi vizibile prin prefixarea numelui lor cu numele package-ului „părinte”. Numele pachetului ar putea fi omis prin *importarea* explicită a claselor sau a întregului package, obligativitatea prefixării rămânând obligatorie doar în cazul unor conflicte potenţiale cu celelalte clase existente.

Cuvântul cheie **import** declară includerea întregului spaţiu de nume (*) sau doar a unor elemente (clase) specificate distinct din spaţiul de nume al package-ului (bibliotecii) indicat(e) astfel. De exemplu pentru importul elementelor bibliotecii *util* din distribuţia Java:

```
import java.util.*
```

Altfel clasele care sunt utilizate de către componenta respectivă vor trebui specificate prin numele lor întreg, adică inclusiv numele complet al bibliotecii (package-ului) acesteia.

```
java.util.Random
```

Codul sursă scris în Java este organizat sub forma unor fişiere cu extensia *.java*. Un astfel de fişier formează ceea ce se mai numeşte o *unitate de compilare*. Într-un fişier *.java* poate fi declarată **public** cel mult o singură clasă care va da obligatoriu şi numele fişierului. Celelalte clase ar trebui să formeze contextul de implementare al clasei *publice* şi sunt vizibile în întregul pachet din care face parte aceasta.

Compilarea unui fişier *.java* generează un alt fişier cu extensia *.class pentru fiecare clasă din fişierul .java*. Fişierele *.class* pot fi împachetate şi arhivate într-o arhivă *JAR*, de unde vor fi localizate şi încărcate de către mediul *runtime* Java.

Împachetarea în aceeaşi unitate (al cărei principal rol este să formeze un spaţiu de nume) a componentelor (asimilând aici unităţile de compilare desemnate prin fişierele *.java* care conţin o clasă publică) se realizează prin intermediul instrucţiunii ***package*** care trebuie plasată pe prima linie necomentată din fişierul sursă şi care desemnează un nume de bibliotecă (package). Clasele *publice* împachetate astfel vor putea fi accesate din exteriorul package-ului sau vor putea fi aduse într-un spaţiu de nume local prin intermediul instrucţiunii ***import***.

Crearea package-urilor având nume unice

Un package nu reprezintă un singur fişier ci, de fapt, este format din mai multe fişiere cu extensia *.class*. Prin urmare, la nivel fizic, organizarea unui astfel de pachet implică organizarea fişierelor *.class*, motiv pentru care toate aceste fişiere trebuie grupate într-un director comun. În acest fel pentru organizarea lor ne vom folosi de structura ierarhică de directoare a sistemului de fişiere. Respectivul director fizic va purta numele *package*-ului de la nivel logic.

Pentru a asigura un nume unic pentru un package, regula de formare recomandată se bazează pe adresele unice din internet (inversate) la care se adaugă numele directorului package-ului. Calea (din sistemul de fişiere) directorului package-ului este inclusă fie într-o variabila de mediu a sistemului de operare, numită *CLASSPATH*, fie în parametrul *–classpath* al mediului runtime. Astfel, în cazul în care:

- calea fizică din sistemul de fişiere este: C:\Java\Clase\exemple\vizibilitate şi
- *classpath* este C:\Java\Clase

atunci numele package-ului poate fi *exemple.vizibilitate*, iar prin instrucțiunea

```
import exemple.vizibilitate.*
```

pot fi invocate prin numele lor toate clasele din respectivul package oriunde s-ar scrie această instrucțiune.

Sau, respectând convenția privind specificarea domeniului internet (*ora.es.uaic.ro*), în cazul în care:

- calea fizică din sistemul de fişiere este: C:\Java\Clase**ro\uaic\es\ora**\exemple\vizibilitate,
- iar *classpath* este C:\Java\Clase

atunci numele package-ului poate fi *ro.uaic.es.ora.exemple.vizibilitate*

Există şi situații în care s-ar putea produce totuşi *coliziuni* de nume. Acest lucru se poate întâmpla în situația în care două clase cu acelaşi nume sunt incluse în package-uri diferite care ar putea fi importate în întregime în aceeaşi locație, sau dacă una din clasele package-ului importat are acelaşi nume cu o clasă existentă în contextul în care se face importul. Pentru a indica exact care dintre aceste clase este invocată, ele vor trebui prefixate cu numele package-ului din care fac parte.

Specificatorii de acces sunt folosiți în interiorul unei *biblioteci* Java [mai exact în cadrul unui package] pentru a avea controlul asupra claselor definite în acea bibliotecă.

Astfel, într-un package o clasă poate fi însoțită de cuvântul cheie **public** pentru a specifica faptul că va fi accesebilă (vizibilă) la ***importul*** package-ului de către o altă clasă definită într-un alt package. Clasele din interiorul aceluiaşi package se pot invoca între ele beneficiind de accesul ***friedly*** specific package-urilor în general, aşa încât nu sunt necesare declarații *public* decât pentru a marca accesul din exteriorul package-ului respectiv.

De reținut că într-o unitate de compilare (fişier *.java*) nu poate fi declarată decât o singură clasă drept *public*ă care va da, de altfel, şi numele fişierului respectiv. Clasele nedeclarate

*public*e sunt disponibile prin acces *friendly* pentru toate clasele definite în package-ul respectiv (nu în mod necesar în aceeaşi unitate de compilare).

Specificatorii *protected* sau *private* nu pot fi folosiți în cazul claselor grupate prin intermediul package-urilor. Dacă se intenționează ascunderea completă a unei clase, atunci se poate opta pentru desemnarea tuturor membrilor din clasa respectivă ca *private*. Oricum clasele nedeclarate *public*-e în package-ul din care fac parte nu pot fi invocate din exterior chiar dacă package-ul este importat în totalitate (*).

Dacă în variabila CLASSPATH este specificat caracterul ‚.' atunci între clasele din directorul curent, chiar dacă nu sunt împachetate împreună printr-o instrucțiune *package*, se formează relații „friendly", mediul Java considerându-le ca făcând parte din ceea ce se numeşte **package-ul *default***.

În exemplul de mai jos este prezentată o situație în care:

- există un package *clienți* incluzând clasele *Companie* şi *Persoana* publice;
- există un package *Bănci* conținând clasele *Banca* şi *Cont*, în care sunt importate clasele publice din package-ul *clienți.* Pe baza clasei *Companie* este construită clasa *Banca*, iar pe baza claselor *Companie* şi *Persoana* sunt declarați membrii *clientiPJuridice* şi *clientiPFizice* ai clasei *Banca*.

```
// package-ul Clienti – fişierul Companie.java
package clienti;
import Salariati.Departament;;

public class Companie {
  public String codFiscal;
  public String nume;
  public String sediu;
  public Departament[] departamente;

// constructorul default
  public Companie(){}
// primul constructor parametrizat
  public Companie(String pCodFiscal, String pNume, String pSediu){}
// al doilea constructor parametrizat
  public Companie(String pCodFiscal, String pNume, String pSediu,
int nrDepartamente){}
}

//------------------------------------------------------------------------------
// package-ul clienti – fişierul Persoana.java
package clienti;
import banci.Cont;
import banci.Banca;

public class Persoana {
  public String cnp;
  public String numePren;
  public String domiciliu;
  protected Cont contBanca;

  protected Persoana(){}
  public Persoana(String pCnp, String pNumePren, String pDomiciliu){
```

```
    cnp = pCnp;
    numePren = pNumePren;
    domiciliu = pDomiciliu;
  }
  public void stabContBanca(String pNrCont, Banca pBanca){
    contBanca = new Cont();
    contBanca.nrCont = pNrCont;
    contBanca.banca = pBanca;
    contBanca.tipCont = new String("PersoanaFizica");
    contBanca.titularCont = this.numePren;
  }
}

//-----------------------------------------------------------------------------
// package-ul banci – fişierul Cont.java
package banci;


public class Cont{
  public String nrCont;
  public Banca banca;
  public String tipClient;
  public String titularCont;
  public String tipCont;

  private float debitCont;
  private float creditCont;
  private float limitaCreditare;
  private float disponibil;
  private float soldCont;

  public float depune(float sumaDepusa){}
  public float retrage(float sumaRetrasa){}
  public void stabLimitaCreditare (float suma) {}
}

//-----------------------------------------------------------------------------
// package-ul banci – fişierul Banca.java
package banci;
import clienti.Companie;
import Salariati.Departament;
import Salariati.Persoana;

public class Banca extends Companie{
  public String abreviereNume;
  public Companie [] clientiPJuridice;
  public Persoana [] clientiPFizice;

  public Banca(String pCod, String pNume, String pIndicativ){
    super(pCod, pNume, "");
    abreviereNume = pIndicativ;
  }
}
```

2.2 Compunerea, moştenire, polimorfism

Productivitatea în programarea orientată obiect provine din maniera de a crea noile clase pe baza celor existente, revalorificând funcţionalitatea acestora. Programele rezultate astfel vor putea fi construite pe baza componentelor existente, testate, documentate şi portabile.

2.2.1 Compunerea

Sintaxa pentru compunerea claselor nu comportă caracteristici speciale şi constă în simpla plasare de referinţe pentru obiecte în interiorul noilor clase, rezultând astfel membri (câmpuri) de tipul unor clase existente.

După cum am menţionat mai înainte, iniţializarea membrilor primitivi se face automat la valori default care sunt, de obicei, echivalente cu 0. Membrii declaraţi însă ca ***referinţe*** de obiecte sunt iniţializaţi implicit la valori *null*, iar metodele lor nu pot fi apelate în această situaţie. Prin urmare membrii-referinţe care compun o clasă nu pot fi accesaţi fără o iniţializare explicită. Aceasta se poate face în trei moduri:

1. În momentul definirii obiectelor membri în clasa în care sunt declaraţi.
2. În constructorul clasei, astfel că instanţierea clasei compozit produce şi iniţializarea membrilor – referinţe.
3. Chiar înaintea folosirii efective a membrilor-referinţe, modalitate numită adesea *iniţializare întârziată*. Astfel s-ar putea reduce fluxul de execuţie suplimentar în situaţiile în care obiectul nu trebuie creat întotdeauna.

```
// -> fisierul Cont.java
public class Cont{
  public String nrCont;
  public String banca;
  //inițializare în momentul definirii
  public String tipClient = new String("Persoana Fizica");
  public String titularCont;
  //inițializare în momentul definirii
  public String tipCont = new String("Debit");

  private float debitCont;
  private float creditCont;
  private float limitaCreditare;
  private float disponibil;
  private float soldCont;

  public float depune(float sumaDepusa){}
  public float retrage(float sumaRetrasa){}
  public void stabLimitaCreditare (float suma) {}
}

// -> fisierul Persoana.java
public class Persoana {
  //inițializare în constructor
  public String cnp;
  public String numePren;
  public String domiciliu;
  protected Persoana(){
  }
  public Persoana(String pCnp, String pNumePren, String pDomiciliu){
    cnp = pCnp;
    numePren = pNumePren;
    domiciliu = pDomiciliu;
  }
  public void stabContBanca(String pNrCont, Banca pBanca){
    // inițializare în momentul folosirii
    contBanca = new Cont();
    contBanca.nrCont = pNrCont;
    contBanca.banca = pBanca;
    contBanca.tipCont = new String("PersoanaFizica");
    contBanca.titularCont = this.numePren;
  }
```

Figura 2-10 Inițializarea membrilor care intră în componența claselor

2.2.2 Moştenire şi polimorfism

A abstractiza înseamnă, simplificând, a elimina diferențele. Două lucruri considerate diferite la un nivel foarte concret, pot fi considerate similare sau echivalente la un nivel *mai abstract*.

Factorul cel mai important sau critic îl constituie faptul că o instanță a unei subclase are toate caracteristicile (trăsăturile) *superclasei* (cu excepția celor private), plus altele suplimentare. O *superclasă* exprimă *funcționalitatea comună şi partajabilă* a subclaselor sale.

2.2.2.1 Extensibilitate şi moştenire

Mecanismul furnizat de un limbaj orientat obiect pentru implementarea abstractizării este numit *extinderea claselor (class extension)*.

Moştenirea este mecanismul prin care o clasă posedă în mod automat toate caracteristicile *non-private* ale superclaselor sale.

Un limbaj de programare nu se poate numi (complet) *orientat-obiect* dacă nu oferă suport consistent pentru moştenire. În Java ierarhiile de clase au o rădăcină comună., prin urmare moştenirea este parte integrantă şi a acestui limbaj. De fapt, întotdeauna când se este creată o nouă clasă se foloseşte acest principiu, chiar dacă nu există vreo indicație explicită în acest sens, fiindcă orice clasă este derivată implicit din clasa rădăcină ***Object***.

Sintaxa pentru moştenire este simplă: pentru a reflecta faptul că o clasă este de tipul altei clase (sau că o clasă derivă dintr-o altă clasă) – relația *is (like) a* – atunci după numele clasei şi înainte de a deschide blocul de descriere al acesteia (înainte de prima paranteză acoladă "{") se menționează cuvântul cheie ***extends*** urmat de numele clasei de bază. Prin această sintaxă noua clasă va avea la dispoziție toți membrii şi metodele *vizibile* din clasa de bază. Această sintagmă reflectă faptul că prin clasa părinte se *generalizează* subclasele, sau că prin subclase se *specializează* superclasa.

2.2.2.2 Generalizare şi subtipizare

Un aspect esențial al generalizării este redat de faptul că, din moment ce o *subclasă are aceeaşi natură ca şi superclasa sa* (de exemplu un *Salariat* este o – is a – *Persoană*), referința la subclasa respectivă conține, de asemenea, şi o referință la superclasa sa (o referință la un *Salariat* este de asemenea o referință la o *Persoană*). În această lumină următoarea secvență de cod este corectă (ştiind că *Salariat*-ul este derivat din subclasa *Persoana*) :

```
Salariat s = new Salariat() ;
Persoana p ;
p = s ; // instrucțiune corectă din moment ce
      //un Salariat este o Persoană
```

Există situații în care este necesară determinarea tipului exact al obiectului la care se face referire printr-o anumită variabilă. În acest sens poate fi utilizat operatorul ***instanceOf***.

De exemplu :

```
s instanceof Persoana
```

va întoarce *true* dacă valoarea variabilei *s* desemnează o *Persoană* sau o subclasă a acesteia (adică un *Salariat*).

Ştiind faptul că o variabilă face referire la o anumită clasă atunci putem aplica o operațiune numită ***cast*** asupra respectivei variabile, pentru a o trata exact ca fiind de tipul clasei a cărei instanță ştim cu siguranță că este grație operatorului *instanceOf* , printr-o sintaxă de genul :

```
(Clasa) reference_value
```

De exemplu:

```
If (s instanceof Persoana)
   ((Persoana)s).stabContBanca() ;
/*sau*/
Persoana p = (Persoana)s ;
p.stabContBanca() ;
```

Exemplu de mai sus a fost posibil datorită principiului numit ***subtipizare*** care stabileşte faptul că *reprezintă mecanismul prin care o instanţă a unei clase poate fi utilizată în orice context în care este specificată superclasa sa.*

2.2.2.3 Clase abstracte şi metode abstracte

Ca efect al generalizării, există situaţii în care o clasă nu este creată în mod necesar pentru a fi instanţiată direct, motiv pentru care ar trebui declarată ca *abstractă*. Prin urmare, scopul ei este de a servi drept fundaţie pentru construirea subclaselor, permiţându-se astfel ca obiecte diferite, dar care se aseamănă între ele în anumite privinţe, să fie manipulate într-o manieră unitară, prin intermediul specificaţiilor superclaselor comune. Invers, o clasă prin care sunt create nemijlocit instanţe este numită *concretă*.

În această privinţă, în limbajul Java se utilizează pentru a face diferenţa între clasele fără posibilităţi directe de instanţiere şi celelalte, cuvântul cheie ***abstract*** :

```
public abstract class BusinessUnit {}
public class Departament extends BusinessUnit{}
```

Deşi clasele abstracte nu pot fi instanţiate, pot fi însă definite variabile care să facă referire la astfel de clase (să fie de tipul acestor clase). După cum am văzut mai înainte, o *referinţă-de-Salariat* este un **subtip** de *referinţă-de-Persoană*. Similar, o *referinţă-de-Departament* este un **subtip** de *referinţă-de-BusinessUnit* :

```
BusinessUnit b;
b = new BusinessUnit() // generează eroare de compilare
b = new Departament() // merge fără probleme
```

Clasele abstracte pot conţine ***metode abstracte***. O metodă abstractă este o metodă specificată dar neimplementată. Datorită faptului că o clasă concretă nu poate conţine o metodă abstractă înseamnă că orice clasă concretă trebuie să asigure implementarea metodelor abstracte specificate în superclasele abstracte. Specificaţia unei metode abstracte se face în felul următor :

```
public abstract class BusinessUnit {
    public abstract BusinessUnit getUnitateSuperioara();
}
```

Se observă lipsa simbolurilor de marcare ale blocului de instrucţiuni executabile {} care ar trebui să formeze corpul metodei, însă (sub)*clasele concrete sunt obligate să implementeze* această metodă :

```
public class Departament extends BusinessUnit{
    private String Nume;
    private Departament unitateSuperioara ;
    public BusinessUnit getUnitateSuperioara(){
        return this.unitateSuperioara;
    };
}
```

În cazul unei variabile de genul :

```
BusinessUnit b = new Departament();
```

dacă *b* este de fapt un Departament atunci pentru apelul *getUnitateSuperioară()* :

```
BusinessUnit depSup;
depSup = b.getUnitateSuperioara();
```

va fi executată metoda definită la nivelul clasei concrete *Departament*. Aceste mecanism este numit *legare întârziată sau dinamică (dynamic binding)* (vezi paragrafele următoare).

2.2.2.4 Constructori şi subclase

Constructorii nu sunt metode moştenite, astfel că fiecare clasă trebuie să-şi definească proprii constructori. Constructorii din subclase sunt responsabili pentru inițializarea variabilelor non-statice (de instanță) definite în superclasă ca şi cei definiți în mod expres la nivelul lor. Prin urmare *primul lucru care trebuie să aibă loc este invocarea fie implicită, fie explicită a constructorului default (fără argumente) din clasa sa părinte* sau invocarea explicită a unui alt constructor din aceeaşi clasă. În oricare din cazuri constructorul clasei părinte trebui totuşi invocat în cele din urmă.

Prin urmare *instanțierea unei clase derivate determină în mod implicit şi instanțierea clasei de bază* (sau a claselor de bază dacă respectiva subclasă se găseşte pe o ramură a unei ierarhii de generalizare care conține mai multe noduri). Deci *constructorul clasei derivate invocă automat şi constructorul din clasa de bază*.

Pentru invocarea explicită a constructorului din clasa părinte este folosit cuvântul cheie ***super*** cu formatul :

```
super(argumente) ;
```

De exemplu pentru clasa *Salariat* derivată din clasa *Persoana* :

```
public class Salariat extends Persoana{
    ... ...
    public Salariat(String pCnp, String pNumePren,
    String pMarca){
        super(pCnp, pNumePren, null) ;
        marca = pMarca ;
    }
}
```

Sau în cazul în care mai există un constructor la nivelul clasei Salariat :

```
public class Salariat extends Persoana{
    ... ...
    public Salariat(String pCnp, String pNume, String pMarca){
        super(pCnp, pNume, null) ;
        marca = pMarca ;
    }
  public Salariat(String pCnp, String pNumePren, String pMarca,
   double pSalTarifar, double pVechime, double pConducere){
   this(pCnp, pNumePren, pMarca);
   marca = pMarca;
   salTarifar = pSalTarifar;
   stabVechime(true, pVechime);
   stabConducere(true, pConducere);
  }
}
```

Dacă este definit un constructor în care nu se face apel explicit la constructorul clasei părinte, atunci se consideră că acest apel se face în mod implicit, ca şi cum prima instrucţiune executabilă ar fi:

```
super();
```

Prin urmare în cazul în care în subclase constructorii acestora fac apel implicit la constructorii din superclase, (sau dacă nu este definit explicit nici un constructor, instanţierea făcându-se prin constructorul implicit furnizat de mediul Java) atunci în respectivele clase de bază trebuie să existe, pe lângă ceilalţi constructori parametrizaţi, şi definiţia constructorului *default* (constructorul fără argumente).

Cuvintele cheie *this* şi *super* folosite pentru invocarea constructorilor din aceeaşi clasă sau din superclase *vor apare întotdeauna pe prima linie din constructorul în care sunt invocaţi.*

2.2.2.5 Suprascrierea şi polimorfismul

După cum am remarcat mai înainte fiecare subclasă moşteneşte în întregime toate caracteristicile (publice) ale clasei părinte. De exemplu, fiecare clasă Java posedă metoda *equals* în virtutea faptului că oricare ar fi, ea are la bază clasa *Object* din ierarhia distribuţiei mediului Java. Codul original al acestei clase este următorul :

```
public boolean equals (Object obj) {
    return this = = obj ;
}
```

Orice subclasă poate redefini o metodă moştenită de la o superclasă din ierarhia de generalizare. O astfel de redefinire se numeşte *rescriere*[4] – overriding. Condiţia este ca noua metodă obţinută prin redefinire să respecte întocmai sintaxa metodei originale, adică să specifice acelaşi tip returnat şi să primească aceleaşi tipuri de argumente şi în aceeaşi ordine. De exemplu putem redefini metoda *equals* pentru clasa Persoana după cum urmează :

```
public class Persoana {
    ... ...
    public boolean equals (Object o){
    if (o instanceof Persoana)
        return this.cnp == ((Persoana)o).cnp ;
    else
        return false ;
    }
}
```

O clasă care redefineşte anumite metode moştenite de la antecedenţii săi poate apela respectivele metode suprascrise (aşa cum sunt ele definite la nivelul superclaselor) prin intermediul cuvântului ***super***, care va însemna de data aceasta o referinţă la clasa de bază. De exemplu pentru clasa Salariat derivată din clasa Persoana metoda *equals* poate fi suprascrisă în felul următor :

```
public class Salariat extends Persoana {
    ... ...
```

[4] Traducerea este destul de aproximativă, dar încearcă să sugereze fenomenul real care are loc

```
    public boolean equals (Object s){
        if (s instanceOf Salariat)
                return super.equals(s) && this.marca == ((Salariat)s).marca;
        else
                return false ;
    }
}
```

În aceeaşi ordine de idei, pentru clasa *SalariatAcord* care redefineşte metoda *calculSalariu*() moştenită din clasa sa de bază Salariat cuvântul *super* poate fi folosit şi după cum urmează :

```
public class SalariatAcord extends Salariat{
  public double pct_Realizat;
  public double calculSalariu() {
    salariu = super.calculSalariu()*pct_Realizat;
    return salariu;
  }
}
```

Prin urmare ***suprascrierea*** (overriding) reprezintă o implementare alternativă pentru o metodă moştenită, iar ***polimorfismul*** reprezintă comportamentul dinamic prin care metoda executată ca urmare unui apel către un obiect dat este determinată la *runtime* luând în considerare clasa din care a fost instanțiat obiectului respectiv.

Supraîncărcarea metodelor pare într-un anume fel asemănătoare cu suprascrierea, însă reprezintă în fapt un lucru destul de diferit. *Supraîncărcarea* permite ca o clasă să posede metode diferite dar cu acelaşi nume atât timp cât ele se deosebesc prin numărul, ordinea sau tipul parametrilor. De exemplu (pentru clasa *Salariat*):

```
public double calculSalariu() {
    ... ...
    return salariu ;
};
public double calculSalariu(double pVechime) {
 stabVechime(true, pVechime) ;
 salariu = (1 – pct_Penalizare)*(salTarifar + sporVechime + sporConducere) ;
 return salariu;
};
public double calculSalariu(double pVechime,
    double pPenalizare) {
    pct_Penalizare = pPenalizare ;
    this.calculSalariu(pVechime) ;
    return salariu;
};

//sau chiar constructori :
public Salariat() ;
public Salariat(String pCnp, String pNume, String pMarca) ;
public Salariat(String pCnp, String pNume, String pMarca, double pVechime, double pConducere) ;
```

Există situații în care se poate face uşor confuzie între ***suprascriere*** şi ***supraîncărcare***. Să luăm în considerare exemplul anterior în care am (re)definit metoda *equals(Object o)* în clasa *Persoana*. Să considerăm apoi că în clasa *Salariat* această metodă este redefinită după cum urmează:

```
public class Salariat extends Persoana {
    public boolean equals (Persoana s){
        if (s instanceof Salariat)
                return super.equals(s) && this.marca = = ((Salariat)s).marca;
        else
                return false ;
    }
}
```

La prima vedere nu există diferențe față de acelaşi exemplu folosit în cazul suprascrierii. Şi totuşi, faptul că această metodă primeşte ca argument un obiect de tip *Persoana* şi nu unul de tip *Object*, modifică în mod esențial datele problemei. Adică pentru clasa Salariat pot fi invocate după metode *equals* :

```
public boolean equals (Object s) ;
public boolean equals (Persoana s) ;
```

Prima metodă cu numele *equals* este moştenită de la clasa Persoana, iar cealaltă este definită la nivelul ei. Prin urmare este vorba de fapt de o *supraîncărcare* şi nu de o *suprascriere*. Să luăm în considerare clasele *Persoana* şi *Salariat* definite ținând cont de toate problemele discutate anterior, vezi figura 3.11

```
//-> fisierul Persoana.java
public class Persoana {
  //inițializare în constructor
  public String cnp;
  public String numePren;
  public String domiciliu;
  public Cont contBanca;
  protected Persoana(){
  }
  public Persoana(String pCnp, String pNumePren, String pDomiciliu){
    cnp = pCnp;
    numePren = pNumePren;
    domiciliu = pDomiciliu;
  }
  public void stabContBanca(String pNrCont, String pBanca){
    // inițializare în momentul folosirii
    contBanca = new Cont();
    contBanca.nrCont = pNrCont;
    contBanca.banca = pBanca;
    contBanca.tipCont = new String("PersoanaFizica");
    contBanca.titularCont = this.numePren;
  }
  public boolean equals (Object o){
    System.out.println("Apel Persoana.equals(Object)");
    if (o instanceof Persoana)
      return this.cnp == ((Persoana)o).cnp ;
    else
      return false ;
  }
}
```

Figura 2-11 Clasa *Persoana*

```
// -> fisierul Salariat.java
public class Salariat extends Persoana{
  public String marca;
  public double pct_Vechime;
  public double pct_Penalizare;
  private double salTarifar;
  private double pct_Conducere;
  private double sporVechime;
  private double sporConducere;
  protected double salariu;

  public double calculSalariu() {
    stabVechime(false, 0);
    stabConducere(false, 0);
    salariu = (1-pct_Penalizare) * (salTarifar + sporVechime + sporConducere);
    System.out.println("Salariat.calculSalariu() "+(int)salariu);
    return salariu;
  }
  public void stabVechime(boolean modifica, double pVechime){
    if (modifica)
      pct_Vechime = pVechime;
    sporVechime = salTarifar * pct_Vechime;
  }
  public void stabConducere(boolean modifica, double pConducere){
    if (modifica)
      pct_Conducere = pConducere;
    sporConducere = salTarifar * pct_Conducere;
  }

  public Salariat(String pCnp, String pNume, String pMarca){
    super(pCnp, pNume, null) ;
    marca = pMarca ;
  }

  public Salariat(String pCnp, String pNumePren, String pMarca,
    double pSalTarifar, double pVechime, double pConducere){
    this(pCnp, pNumePren, pMarca);
    salTarifar = pSalTarifar;
    stabVechime(true, pVechime);
    stabConducere(true, pConducere);
  }

  public boolean equals (Persoana s){
    System.out.println("Apel Salariat.equals(Persoana)");
    if (s instanceof Persoana)
      return super.equals(s) && this.marca == ((Salariat)s).marca;
    else
      return false ;
  }
}
```

Figura 2-12 Clasa *Salariat*

Dacă vom executa următoarea clasă *Test*

```
public class Test {
  public static void main(String[] args) {
  Persoana p1 = new Persoana("CNP1", "O persoana", null);
  Salariat s1 = new Salariat("CNP2", "Un salariat", null);
  Object x1 = new Object();

  System.out.println("s1= obiectul x1");
  s1.equals(x1);
  System.out.println("");

  System.out.println("s1= persoana p1");
  s1.equals(p1);
```

```
    System.out.println("");

    System.out.println("s1= salariatul x1");
    s1.equals(s1);
    }
  }
```

vom obţine următorul rezultat:

```
s1= obiectul x1
Apel Persoana.equals(Object)

s1= persoana p1
Apel Salariat.equals(Persoana)
Apel Persoana.equals(Object)

s1= salariatul x1
Apel Salariat.equals(Persoana)
Apel Persoana.equals(Object)
```

Acest rezultat arăta clar cum pentru obiectul s1 de tip *Salariat* au fost apelate două metode cu numele *equals* diferite, şi, mai mult, una dintre ele o apelează intern pe cealaltă.

2.2.2.6 Utilizarea moştenirii

Moştenirea, deşi reprezintă un concept mai dificil şi mai complex, poate constitui un puternic instrument pentru a construi sisteme flexibile din componente reutilizabile. Există mai multe situaţii în care moştenirea poate constitui o cale pentru structurarea unor soluţii elegante. Astfel :

- *Unificarea funcţionalităţii comune între clase.* În acest caz moştenirea constituie suportul abstractizării când dorim să tratăm în acelaşi mod o colecţie de instanţe provenind din clase diferite, exploatând astfel polimorfismul. De exemplu am putea să afişăm numele tuturor persoanelor care lucrează într-o companie indiferent dacă sunt salariaţi interni sau doar colaboratori :

```
public class Persoana {
  public String cnp;
  protected String numePren;
  public String domiciliu;
  public Cont contBanca;

  public String getNume(){
    return numePren;
  }
  public Persoana(String pCnp, String pNumePren, String pDomiciliu){
    cnp = pCnp;
    numePren = pNumePren;
    domiciliu = pDomiciliu;
  }
... ... ...
}

public class Colaborator extends Persoana {
  public String nrConventie;
  public Colaborator(String pCnp, String pNumePren, String pNrConventie){
    super(pCnp, pNumePren, null);
    nrConventie = pNrConventie;
  }
```

```
}
public class Salariat extends Persoana {
  public String marca;
  public Salariat(String pCnp, String pNume, String pMarca){
    super(pCnp, pNume, null) ;
    marca = pMarca ;
  }
... ... ...
}

public class TestPolimorfism {
  public static void main(String[] args) {
    Persoana[] angajati = {new Salariat("CNP1", "Salariat 1", "10001"),
            new Colaborator("CNP2", "Colaborator 1", "20001"),
            new Salariat("CNP2", "Salariat 2", "10002"),
            new Persoana("CNP3", "Persoana 1", null)} ;
    for (int i=0 ; i < angajati.length ; i++ )
      System.out.println(angajati[i].getNume());
  }
}
```

- *Asigurarea implementării pentru o metodă abstractă*. Una din cele mai comune utilizări ale moştenirii este asigurarea implementării metodelor definite în clasele abstracte prin intermediul claselor concrete, pe baza polimorfismului făcându-se legătura efectivă dintre apelul metodei abstracte şi codul executabil efectiv.

```
public abstract class SegmentArbore {
  public abstract SegmentArbore aflaNodSuperior();
}

class Radacina extends SegmentArbore{
  private String Nume;
  public SegmentArbore aflaNodSuperior(){
    return null;
  }
}

class Ramura extends SegmentArbore{
  private String Nume;
  private Ramura nodSuperior ;
  public SegmentArbore aflaNodSuperior(){
    return this.nodSuperior;
  }
}

class Frunza extends SegmentArbore{
  private String Nume;
  private Ramura ramuraSuperioara ;
  public SegmentArbore aflaNodSuperior(){
    return this.ramuraSuperioara;
  }
}
```

- *Rafinarea implementării unei metode.* Atunci când rafinăm o metodă nu facem altceva decât să o suprascriem în aşa fel încât noua versiune de la nivelul subclasei să extindă funcționalitatea inițială sau să țină seama de anumite trăsături caracteristice acesteia. Este cazul concret al metodei *calculSalariu*() care la nivelul clasei SalariatAcord

trebuie să țină seama de procentul realizat funcție de care se calculează drepturile efective.

```
public class Salariat extends Persoana{
  public String marca;
  public double pct_Vechime;
  public double pct_Penalizare;
  private double salTarifar;
  private double pct_Conducere;
  private double sporVechime;
  private double sporConducere;
  protected double salariu;

  .... .... ....
  public double calculSalariu() {
    stabVechime(false, 0);
    stabConducere(false, 0);
    salariu = (1-pct_Penalizare) * (salTarifar + sporVechime + sporConducere);
    System.out.println("Salariat.calculSalariu() "+(int)salariu);
    return salariu;
  }
  ... ... ...
  public void stabVechime(boolean modifica, double pVechime){
    if (modifica)
      pct_Vechime = pVechime;
      sporVechime = salTarifar * pct_Vechime;
  }
  public void stabConducere(boolean modifica, double pConducere){
    if (modifica)
      pct_Conducere = pConducere;
    sporConducere = salTarifar * pct_Conducere;
  }
... ... ...
}

public class SalariatAcord extends Salariat{
  public double pct_Realizat;

  public double calculSalariu() {
    salariu = super.calculSalariu()*pct_Realizat;
    System.out.println("SalariatAcord.calculSalariu() "+ (int)salariu);
    return salariu;
  }
}
```

- *Extinderea funcționalității unei clase existente.* De exemplu, pentru abstractizarea salariaților unei companii poate exista o clasă *Salariat* prin care pot fi reprezentați toți salariații, plus o altă clasă *SalariatAcord* pentru cei care lucrează în acord care au specific lucru pe bază de lucrări. Prin urmare clasa *SalariatAcord* a apărut ca urmare a extinderii specificației clasei Salariat.

2.2.3 Alte concepte legate de abstractizare

2.2.3.1 Clase abstracte

Am văzut mai înainte că o clasă abstractă spre deosebire de clasele concrete poate să conțină metode abstracte adică metode care sunt doar specificate, fără implementare. Altfel, o clasă abstractă este la fel ca orice clasă : poate participa în ierarhii de moştenire în care să fie superclasă sau subclasă, poate avea constructori (deşi niciodată aceste clase nu sunt instanțiate în mod direct) constructori care sunt apelați la instanțierea subclaselor concrete, pot fi definite variabile sau parametri ale căror tipuri să reflecte tocmai aceste clase.

2.2.3.2 Interfețe

În Java o *interfață* este asemănătoare cu o clasă abstractă, însă nu poate conține constructori sau metode având specificate inclusiv codul executabil (o clasă abstractă poate conține şi metode abstracte şi metode complet specificate, restricția fiind ca, dacă exista cel puțin o metodă neimplementată, clasa să fie definită abstractă) şi nici variabile. Prin urmare în declarația unei interfețe nu există *nici o " urmă" de implementare*. O interfață este folosită doar ca un view limitat asupra unui grup de obiecte, sau pentru a specifica un set minimal de caracteristici care ar trebui să se găsească la un anumit grup de de obiecte ce ar trebui să satisfacă o cerință comună sau să realizeze un scop comun.

Prin urmare, definiția unei interfețe poate conține numai specificațiile metodelor (metodele abstracte) şi definiții de constante. La fel ca o clasă abstractă, o interfață nu poate fi instanțiată. De asemenea, tot la fel ca şi o clasă abstractă, o interfață poate fi inclusă într-un package. De exemplu package-ul *java.awt* conține o interfață numită *LayoutManager* definită în felul următor :

```
public interface LayoutManager {
    Dimension minimumLayoutSize (Container parent) ;
    Dimension preferredLayoutSize (Container parent) ;
    void addLayoutComponent (String name, Component comp) ;
    void removeLayoutComponent (Component comp) ;
    void layoutContainer (Container parent) ;
}
```

Datorită faptului că toate metodele şi constantele definite într-o interfață sunt publice în mod implicit, iar metodele sunt în mod implicit abstracte, declarațiile **public** sau **public abstract** nu sunt necesare.

2.2.3.3 Implementarea interfețelor

O clasă *implementează - implements* o interfață în mare parte la fel cum o clasă abstractă este *extinsă - extends* de subclasale sale concrete. Clasa care implementează o interfață este obligată să implementeze absolut toate metodele specificate în iterfată. De exemplu instrucțiunea

```
public class FlowLayout implements LayoutManager {…}
```

cere ca metodele *minimimLayoutSize*, *preferredLayoutSize* etc. să fie implementate în clasa *FlowLayout* exact aşa cum sunt specificate în interfaţă.

Nu este de ajuns ca o clasă să implementeze întâmplător toate metodele unei interfeţe. Faptul că o clasă se conformează unei interfeţe trebuie declarat în mod explicit prin intermediul cuvântului cheie ***implements***.

Dacă o superclasă implementează o interfaţă atunci aceasta este implementată de către fiecare dintre subclasele sale în mod implicit. Prin urmare *relaţia de implementare este tranzitivă*.

O interfaţă poate extinde o altă interfaţă foarte asemănător cu modul în care o clasă extinde o altă clasă. Relaţia dintre interfeţe este de *extidere – extends* pe când relaţia dintre clase şi interfeţe este de *implementare – implements*.

```
public interface LayoutManager2 extends LayoutManager{
    void addLayoutComponent (Component comp, Object constraints) ;
    Dimension maximumLayoutSize (Container target) ;
    float getLayoutAlignamentX (Container target) ;
    float getLayoutAlignamentY (Container target) ;
    void invalidateLayout (Container target) ;
}
```

Orice clasă care implementează interfaţa *LayoutManager2* trebui să implementeze cele cinci metode din declaraţia acesteia plus cele cinci din definiţie interfeţei *LayoutManager*.

Pentru un exemplu mai explicit să considerăm o porţiune dintr-o aplicaţie privind gestiunea conturilor bancare în care avem definite clasele *Banca* şi *Cont*. Titularii conturilor sunt clienţii băncii care sunt fie persoane fizice fie persoane juridice. Pentru specificarea titularului unui cont se poate folosi un tip care să desemneze de fapt o interfaţă la care să se conformeze obiecte din alte aplicaţii care în privinţa relaţiei cu băncile sunt clenţii ai acestora. De exemplu :

```
//******************************************************************************
// contextul aplicaţiei de gestiune a conturilor
package banci;

class Banca {
  String numeBanca ;
  String idBanca ;
  String abreviereNume ;
  String sediu ;
}

public class Cont {
  String nrCont ;
  Banca banca ;
  Client titularCont ;
  String numeTitular ;
  public Cont(Banca pBanca, Client pTitular) {}
}
//-----------------------------------------------------------------------------------

package banci;
public interface Client {
  Cont deschideCont(String pBanca, String pNume, Client pidClient);
  String pidClient();
  String aflaNume();
  String aflaID();
```

```
}


//*****************************************************************************
// contextul unei aplicatii care apelează
// la gestiunea conturilor

package clienti;
import banci.*;

public class Persoana implements Client {
  String nume ;
  String cnp ;
  Cont contBancar ;
  String pid;

  public Cont deschideCont(String pBanca, String pNume, Client pidClient) {
    contBancar  = new Cont(pBanca, pNume, this) ;
    return contBancar;
  }
  public String pidClient(){
    return pid;
  }
  public String aflaNume(){
    return nume;
  }
  public String aflaID(){
    return cnp;
  }
  public Persoana(String pCnp, String pNume){
    cnp = pCnp;
    nume = pNume;
  }
}
```

2.2.3.4 Moştenire multiplă

Deosebirea esențială dintre interfețe şi ierarhiile de clase rezidă în faptul că, în cazul primelor, acestea pot extinde mai mult decât un singur părinte. Acest lucru este uneori numit moştenire multiplă. De exemplu fiind date interfețele *DataInput* şi *DataOutput* poate fi definită o interfață *DataIO* după cum urmează :

```
public interface DataIO extends DataInput, DataOutput{
… ... ....
}
```

Această interfață include atât specificațiile rezultate din *DataInput* cât şi pe cele rezultate din *DataOutput*. Dacă există o clasă care implementează *DataIO* va trebui să conțină implementările specificațiilor atât din *DataInput* cât şi din *DataOutput*.

O importanță mai mare o are însă faptul că o clasă în sine poate implementa în mod direct mai multe interfețe. Adică ar putea avea mai mulți părinți interfețe în afară eventual de unul singur care ar putea fi o clasă obişnuită. Aceste lucru am putea spune că se manifestă implicit pentru orice clasă care implementează o interfață din moment ce aceasta moşteneşte implicit şi clasa *Object*.

2.2.3.5 Referințe la tipuri-interfețe

O interfață la fel ca şi o clasă poate defini un tip la care se fac referințe. Astfel chiar dacă nu putem crea în mod direct interfețe putem crea variabile de tip *referință-LayoutManager* :

```
private LayoutManager mgr ;
```

Această variabilă poate să facă referire la o instanță a oricărei clase care implementează respectiva interfață :

```
mgr = new FlowLayout();
```

Regulile de subtipizare care se aplică claselor se aplică în aceeaşi măsură şi interfețelor.

La fel, în cazul schiței aplicației de gestiune a conturilor dintr-unul din exemplele de mai sus, membrul *titularCont* este de tip *Client* care nu este altceva decât o interfață. De asemenea unul din parametrii constructorului *Cont()* este de tip *Client*, iar în momentul invocării acestuia primeşte o referință la un obiect instanțiat din clasa *Persoana* care implementează interfața *Client*.

2.2.3.6 Avantajele interfețelor față de clasele abstracte

- Interfețele sunt prin definiție abstracte; ele nu au în vedere nici un aspect privind implementarea ;
- O clasă poate implementa mai mult decât o interfață ;
- Interfețele permit o folosire într-o formă mult mai generalizată a polimorfismului, în care instanțele unor clase aparent fără legătură pot fi tratate în mod identic în anumite scopuri.

2.2.3.7 Moştenirea şi accesibilitatea membrilor - Accesebilitate protected

După cum am văzut mai înainte, specificatorii de acces caracteristici mediilor Java sunt *public* pentru acces nerestricționat din afara contextului clasei, *private* pentru acces restricționat numai la contextul intern al clasei, "*friendly*" pentru acces implicit la membrii necalificați printr-o directivă explicită de acces doar în contextul package-ului în care este definită clasa şi *protected*, specificator de acces care nu are sens decât în contextul unor relații de moştenire.

Ca urmare, membrii publici ai unei clase, indiferent de existența vreunei relații de moştenire, sunt accesibili din oricare clasă sau obiect (bineînțeles clasele din afara package-ului în care se găseşte clasa proprietară a membrilor accesați trebuie să declare explicit vizibilitatea la

contextul package-ului prin intermediul intrucțiunii *import*). Pe de altă parte membrii privați nu sunt accesibili în nici un fel înafara clasei de care aparțin, deci nici în subclasele acesteia. Prin urmare, *o subclasă va moşteni* în mod implicit de la clasa sa de bază *numai membrii publici*. Atunci când se doreşte însă ca o parte din membri să rămână inaccesibili din afară, ca în cazul membrilor privați, dar contextul lor de vizibilitate să fie extins totuşi la nivelul subclaselor dintr-o ierarhie de moştenire se foloseşte accesul *protected*.

Prin urmare relațiile de moştenire determină accesul la membrii calificați ca *protected*.

Vizibilitatea poate fi simplu exprimată printr-un exemplu de genul următor:

```
class Super {
    public int public_Super_Field;
    protected int protected_Super_Field;
    private int private_Super_Field;

    public Super(){
        public_Super_Field = 10;
        protected_Super_Field = 20;
        private_Super_Field = 30;
    }
}

class Sub extends Super{
    public int public_Sub_Field;
    protected int protected_Sub_Field;
    private int private_Sub_Field;
    public Sub(){
        public_Sub_Field = 100;
        protected_Sub_Field = 200;
        private_Sub_Field = 300;
    }
}
```

Pe baza acestor clase pot fi încercate testele următoare:

```
class Client {
  public void test() {
    Super mySuper = new Super();
    Sub mySub = new Sub();

    // primul test - toate cele trei variabile sunt valide
    int i = mySuper.public_Super_Field ;
    // prin moştenire de la Super
    int j = mySub.public_Super_Field ;
    int k = mySub.public_Sub_Field ;
    // al doilea test - nici o variabilă nu este valabilă
    int l = mySuper.private_Super_Field ;
    int m = mySub.private_Super_Field ;
    int n = mySub.private_Sub_Field ;
    // al treilea test - nici o variabilă nu este valabilă
    int o = mySuper.protected_Super_Field ;
    int p = mySub.protected_Super_Field ;
    int r = mySub.protected_Sub_Field ;
  }
}
```

Declarația variabilei *j* din primul test este valabilă pentru că grație moștenirii membrul *public_Super_Field* este public şi disponibil şi în subclasele clasei *Super*. Al doilea test este invalid datorită faptului că accesul la membrii privați nu este permis în afara clasei în care sunt declarați. Al treilea test este invalid pentru că membrii protejați sunt accesibili numai în clasa în care sunt declarați şi în clasele care au în mod specific relații de moştenire cu clasa în care sunt declarați.

Ce se întâmplă însă cu accesul membrilor unui obiect din punctul de vedere al altui obiect din aceeaşi clasă. Regula este ca *dacă un membru X, fie moştenit sau definit într-o clasă, este accesibil într-o instanță din clasa respectivă, atunci va fi de asemenea accesibil din toate instanțele acelei clase.* Pentru exemplificare să adăugăm în clasa *Super* următoare metodă care conține trei declarații de variabile inițializate cu valori ale membrilor unei alte instanțe:

```
class Super {
    …
    public void superToSuper (Super anotherSuper) {
        int i = anotherSuper.public_Super_Field ;
        int j = anotherSuper.protected_Super_Field ;
        int k = anotherSuper.private_Super_Field ;
    }
    …
}
```

Se observă că instanța în care sunt declarate cele trei variabile are acces la toți membrii unei alte instanțe indiferent dacă sunt declarați public, protected sa private.

Să luăm acum în considerare următorul exemplu:

```
class Sub {
    …
    public void subToSub (Sub anotherSub) {
        // pentru membrii direcți
        int i = anotherSub.public_Sub_Field ;
        int j = anotherSub.protected_Sub_Field ;
        int k = anotherSub.private_Sub_Field ;
        // pentru membrii moşteniți
        int l = anotherSub.protected_Super_Field ;
        int m = anotherSub.public_Super_Field ;
        // invalid, membri neaccesibili din clasa de bază
        int n = anotherSub.private_Super_Field ;
    }
    …
}
```

Prin urmare membrii vizibili sunt cei declarați direct la nivelul clasei sau cei moşteniți. Dacă luăm în considerare testul următor vom observa că pentru instanțe din clase diferite numai membrii publici vor fi vizibili:

```
class Super {
    …
    public void superToSub (Sub sub) {
        // i este Ok pentru că este vorba de un membru public
        int i = sub.public_Sub_Field ;
        // j şi k sunt invalide, accesul protected sau private este restricționat
        int j = sub.protected_Sub_Field ;
        int k = sub.private_Sub_Field ;
    }
    …
}
```

```
class Sub {
    …
    public void subToSuper (Super sup) {
        // i este Ok pentru că este vorba de un membru public
        int i = sup.public_Super_Field ;
        // j şi k sunt invalide, accesul protected sau private este restricționat
        int j = sup.protected_Super_Field ;
        int k = sup.private_Super_Field ;
    }
    …
}
```

2.2.3.8 Casting

Operația numită „*casting”* reprezintă în fapt procesul conversiei unui tip de bază în alt tip. Un exemplu banal ar fi următorul:

```
double x = 3.255;
int nx = (int)x;
```

Pentru a inițializa corect variabila *nx* variabila *x* din care este preluată valoarea a trebuit compatibilizată cu tipul variabilei *nx*, adică *int*. Acest lucru a fost realizat prin ***casting***.

La fel ca în exemplul anterior în care ne-am bazat pe tipuri primitive, este posibil să apară şi situații în care se poate dovedi necesară reconversia unui obiect dintr-o clasă în alta. La fel ca şi în cazul tipurilor primitive acest proces de asemenea se numeşte *casting*, sintaxa fiind similară, adică înainte de numele variabilei care face referire la obiectul vizat se specifică între paranteze rotunde tipul țintă care va rezulta.

```
Departament d = new Departament();
d.membri = new Salariat[2];
d.membri[0] = new Salariat();
// posibil prin moştenire:
d.membri[1] = new SalariatAcord();
// casting
Salariat s = (Salariat) d.membri[1];
```

Sau pentru exemplul următor:

```
class Data {
  int an, zi, luna;
  public Data(int an_, int luna_, int zi_){
    an = an_;
    luna = luna_;
    zi = zi_;
  }
  public int getAn(){return an;}
  public int getLuna(){return luna;}
  public int getZi(){return zi;}
}

public class ManagerTest {
  public static void main (String args[]){
    Angajat[] staff = new Angajat[3];
    staff[0] = new Angajat("X", 35000, new Data(1989, 10, 1));
    staff[1] = new Manager("Y", 75000, new Data(1987, 12, 15));
    staff[2] = new Angajat("Z", 85000, new Data(1990, 3, 15));
```

```
    for(int i=0; i<staff.length; i++) staff[i].crestereSalariu(5);
    for(int i=0; i<staff.length; i++) staff[i].print();
  }
}

class Angajat {
  public Angajat (String n, double s, Data d) {
    nume = n;
    salariu = s;
    dataAngajare = d;
  }
  public void print() {
    System.out.println(nume+" "+salariu+" "+ anAngajare());
  }
  public void crestereSalariu(double procent){
    salariu *= 1+procent/100;
  }
  public int anAngajare(){return dataAngajare.getAn();}
  public String nume;
  private double salariu;
  protected Data dataAngajare;
}

class Manager extends Angajat{
  public Manager(String n, double s, Data d){
    super(n, s, d);
    numeSecretara = "";
  }
  public void crestereSalariu(double procent){
    Data astazi = new Data(2003, 03, 31);
    double bonus = 0.5 * (astazi.getAn() - dataAngajare.getAn());
    super.crestereSalariu(procent+bonus);
  }
  public void setNumeSecretara(String n){
    numeSecretara = n;
  }
  public String getNumeSecretara(){
    return numeSecretara ;
  }
  private String numeSecretara;
}
```

Utilizarea operației de *casting* are de regulă o singură motivație - *folosirea unui obiect la întreaga sa capacitate dacă tipul său real a fost reconsiderat*. În exemplul de mai sus, vectorul *staff* ar trebui să fie un tablou de elemente *Angajat* din moment ce o parte sunt doar angajați obişnuiți. Pentru a avea acces la anumiți membri specifici de tip Manager elementele *Angajat* trebuie reconvertite din tipul de bază în tipul *Manager*.

După cum am precizat într-unul din paragrafele anterioare, declarația oricărei variabile în Java trebuie însoțită de tipul ei. Acest tip nu face altceva decât să descrie „genul” de obiect la care se face referire şi ceea ce se aşteaptă să poată face acel obiect. De exemplu *staff[i]* se referă la obiecte *Angajat*, deşi se poate referi, de asemenea, la obiecte *Manager*. Din punctul de vedere al compilatorului se verifică ca programatorul să nu „supra-liciteze” o variabilă la inițializarea ei. Prin urmare, dacă unei variabile de tip *superclasă* îi este asignat un obiect din *subclasă* compilatorul nu „protestează” în nici un fel acceptând necondiționat acest lucru. Dacă însă unei

variabile de tip *subclasă* îi este asignat un obiect din *superclasă* atunci acesta este super-evaluat față de tipul variabilei (se promite mai mult) şi atunci compilatorul cere explicit confirmarea acestui lucru. Această confirmare se face prin notația specică *casting*-ului.

Ce se întâmplă însă dacă se foloseşte operația de *casting* în inițializarea unei variabile, dar nu se respectă "promisiunea" făcută, sau cu alte cuvinte se "minte" la compilare în legătură cu obiectele atribuite efectiv la run-time. De exemplu (bazându-ne pe clasele definite anterior):

```
Manager boss = (Manager)staff[0];
```

La rularea programului, mediul run-time Java observă "promisiunea" nerespectată şi va genera o excepție, execuția oprindu-se brusc la acest moment. Din acest motiv se dovedeşte a fi necesar ca, înainte de a face astfel de operațiuni, obiectul implicat să fie verificat din punctul de vedere al tipului său efectiv (determinat prin momentul instanțierii) care trebuie să corespundă tipului țintă indicat prin casting. În acest caz operatorul *instanceof* se dovedeşte extrem de util:

```
Manager boss;
if (staff[0] instanceof Manager){
    boss = (Manager)staff[0];
    …
}
```

În fine, ceea ce poate face compilatorul este să împiedice declararea castingului în cazul în care acesta nu poate fi realizat în nici o situație, adică atunci când tipul țintă nu este o subclasă a tipului original al obiectului atribuit. De exemplu o instrucțiune de genul

```
Window w = (Window) staff[1];
```

va genera o eroare de compilare din moment ce clasa *Window* nu este derivată din clasa *Angajat*.

2.2.3.9 Cuvântul cheie **final**

Există situații în care se doreşte evitarea sau împiedicarea derivării unei anumite clase de către alte clase. Prin urmare ierarhiile de moştenire ar putea fi încheiate prin clase care să nu accepte derivarea în alte subclase. Astfel de clase se mai numesc clase frunză din perspectiva ierarhiei de moştenire, sau clase *final*-e. Clasele care nu vor fi clase „părinte" sunt calificate prin specificatorul de modificare *final* care însoțeşte declarația clasei respective. Prin urmare antetul unei astfel de clase ar putea arăta astfel:

```
final class Card {... ...}
```

Acelaşi calificator poate fi folosit şi în cazul unei anumite metode. În acest caz prin *final* se declară faptul că respectiva metodă nu mai poate fi suprascrisă (override) în subclasele în care este derivată clasa originală. În acest context trebuie menționat şi faptul că toate metodele dintr-o clasă declarată *final* sunt în mod implicit şi ele *final*-e fără vreo declarație explicită. O clasă sau o metodă poate fi declarată *final* din două motive:

1. *Eficiență*

Legarea dinamică produce mai mult "overhead" (trafic privind informațiile de control) decât cea statică - ca urmare metodele virtuale rulează mai încet. Mecanismul de legare dinamică (dynamic dispatch mechanism) este mai puțin eficient decât apelul direct al unei proceduri. Şi, mai important, compilatorul nu poate înlocui o metodă originală cu o linie de cod directă („inline

code") fiindcă este posibil ca o clasă derivată să suprascrie respectivul cod de apel al metodei. Compilatorul însă poate înlocui apelurile de proceduri cu linii de cod directe dacă acestea sunt declarate *final*.

Apelurile de proceduri nu sunt instrucţiunile "preferate" ale procesoarelor datorită interferării cu strategiile proprii de preluare şi decodare ale instrucţiunilor următoare în timp ce le execută pe cele curente.

2. *Siguranţă*

Flexibilitatea mecanismului de legare dinamică are şi efectul lipsei de control asupra a ceea ce se va întâmpla efectiv la apelul metodelor. Atunci când este expediat un mesaj este posibil ca obiectul căruia îi este adresat să provină dintr-o clasă derivată în care metoda respectivă să fie rescrisă într-o manieră cu totul nouă astfel încât tipul sau structura valorii returnate să fie modificată într-o manieră cu totul neaşteptată. Acest lucru poate fi evitat prin declararea respectivei metode ca *final*.

2.2.3.10 Clase interne – Inner classes

Clasele interne (inner classes) reprezintă un mijloc prin care se pot defini noi clase în interiorul definiţiilor unor clase existente (clasele interne nu sunt acelaşi lucru cu membrii unei clase-compozit). Această facilitate permite gruparea claselor care aparţin aceloraşi structuri logice şi, de asemenea, oferă posibilitatea controlul vizibilităţii unora prin intermediul altora fără a folosi suprastructuri de genul pachetelor (package-urilor).

Metodele clasei în care au fost definite clasele interne le apelează la fel ca oricare alte clase externe. Singura diferenţă este că apelul este rezolvat în spaţiul de nume imediat desemnat de definiţia clasei ce conţine respectivele clase interne. Invocarea claselor interne din exteriorul clasei în care au fost definite presupune:

- în cazul apelurilor *static*-e problema este simplă – clasele interne nu pot avea membri statici;
- în cazul apelurilor non-*static*-e – obiectele sunt create specificând tipul acestora printr-o sintaxă cum ar fi *NumeClasaExternă.NumeClasăInternă*. Obiectelor care au ca tip o clasă internă pot fi create într-o clasă externă în două variante:
  1) apelând constructorul clasei interne (care trebuie declarată *public*), printr-o sintaxă de genul

```
NumeClsExt <ref_cls_ext> = new NumeClsExt() ;
NumeClsExt.NumeClsInt   <ref_cls_int>   =   <ref_cls_ext>.new
NumeClsInt;
```

(atunci când nu există o metodă explicită a clasei exterioare care să invoce constructorul clasei interne).

Ca exemplu să luăm în considerare următoarele definiţii:

```
class ClasaExterna{
  // clasa interna inclusa in ClasaExterna
  class ClasaInterna {
    private String m1 = "Valoare interna";
    // metoda de acces la membrul clasei interne
    public String getm1() {return m1;}
```

```
  }

  // metoda a clasei externe de acces la membrul clasei interne
  public String getMembruIntern(ClasaInterna obj_int){
    return obj_int.getm1();
  }
}

// Clasa de test prin care va fi accesata clasa interna prin
// intermediul clasei externe care o include
public class TestInnerClasses {
  public static void main(String[] args) {
  // creez o instanta a clasei externe
    ClasaExterna o = new ClasaExterna();
 // creez o referinta la clasa interna
    ClasaExterna.ClasaInterna obj_intern = o.new ClasaInterna();
// apelez o metoda a clasei interne
    System.out.println(obj_intern.getm1());
  }
}
```

2) în clasa ce conține clasa internă există o metodă specială care să invoce constructorul clasei interne, metodă care va returna o referință către obiectul creat. Clasele interne nu pot defini membri statici, prin urmare nu poate fi apelată o metodă statică a unei clase interne.

```
NumeClsExt <ref_cls_ext> = new NumeClsExt() ;
NumeClsExt.NumeClsInt <ref_cls_int> =
     <ref_cls_ext>.MtdInstanțiere();
```

```
class ClasaExterna{
  // clasa interna inclusa in ClasaExterna
  class ClasaInterna {
    private String m1 = "Valoare interna";
    // metoda de acces la membrul clasei interne
    public String getm1() {return m1;}
  }
  // metoda pentru instantierea clasei interne
  public ClasaInterna referintaInterna() {
    return new ClasaInterna();
    // sau
    //return new ClasaExterna.ClasaInterna();
  }
  // metoda a clasei externe de acces la membrul clasei interne
  public String getMembruIntern(ClasaInterna obj_int){
    return obj_int.getm1();
  }
}

// Clasa de test prin care va fi accesata clasa interna prin
// intermediul clasei externe care o include
public class TestInnerClasses {
  public static void main(String[] args) {
  // creez o instanta a clasei externe
    ClasaExterna o = new ClasaExterna();
  //  creez o instanta a clasei interne
    ClasaExterna.ClasaInterna obj_intern = o.referintaInterna();
    System.out.println(obj_intern.getm1());
     }
}
```

Clasele interne şi „upcasting"

Clasele interne pot fi folosite pentru implementarea total privată a interfeţelor. Clasele obişnuite nu pot fi *private* sau *protected*, însă clasele interne pot fi declarate *private* sau *protected* în clasa în care sunt definite. Acest fapt are importanţă pentru implementarea complet ascunsă a interfeţelor.

Clasele interne pot implementa interfeţe *publice* şi pot fi declarate *private* sau *protected* în clasa în care sunt definite. Instanţierea acestora se poate face prin metode specifice în clasa în care sunt incluse, metode care returnează referinţe ale obiectelor create din respectivele clasele interne, clase ce implementează interfeţe publice. Prin urmare se pot accesa referinţe la inţerfeţe publice prin intermediul claselor interne care le implementează, ascunse în alte clase. Cu alte cuvinte referinţa la o interfaţă se face printr-un obiect care o implementează complet privat în mod similar cu upcasting-ul de la clasa internă la o clasă de bază. De exemplu, în acelaşi pachet avem definită interfaţa:

```
package testinner;

public interface InterfaceInner {
  String m1 =  "Valoare Interna";
  String getm1();
}
```

şi o secvenţă de clase de test:

```
package testinner;

class ClasaExterna{
  // clasa interna inclusa in ClasaExterna
  private class ClasaInterna implements InterfaceInner {
    private String m1 = "Valoare referinta interna pentru interfata publica";
    // metoda de acces la membrul clasei interne
    public String getm1() {return m1;}
  }
  // metoda pentru instantierea clasei interne
  public ClasaInterna referintaInterna() {
    return new ClasaInterna();
  }
  // metoda a clasei externe de acces la membrul clasei interne
  public String getMembruIntern(ClasaInterna obj_int){
    return obj_int.getm1();
  }
}

public class TestInnerPrivateClasses{
  public static void main(String[] args) {
    ClasaExterna o_ext = new ClasaExterna();
    // Instantierea clasei interne pentru a obtine o referinta tip InterfaceInner
    // nu este posibila datorita accesului privat
    // InterfaceInner i_ref1 = o_ext.new ClasaInterna();

    // Se poate obtine o referinta la interfata InterfaceInner prin intermediul
    // clasei interne folosind o metoda publica din clasa externa a carei
    // responsabilitate este sa instantieze clasa interna
    InterfaceInner i_ref = o_ext.referintaInterna();
     // apelul metodei getm1 definită în InterfaceInner şi implementată în ClasaInternă
     System.out.println(i_ref.getm1());
  }
```

```
}
```

Rezultatul final va fi:

```
Valoare referinta interna pentru interfata publica
```

Declararea unei clase în interiorul unei metode.

O clasă poate fi definită în corpul unei metode aparţinând unei clase obişnuite. Clasa internă va fi compilată împreună cu celelalte clase (obişnuite), însă va putea fi apelată doar în interiorul metodei respective şi nu în afara ei. De asemenea, o clasă internă poate fi definită în interiorul unei structuri de control (de exemplu *if(){}*) şi va putea fi accesată numai în interiorul acestei structuri de control *if(){<spaţiu de accesare>}* şi nu în spaţiul care desemnează corpul metodei mai puţin structura de control respectivă.

Clase interne anonime.

Clasele interne sunt folosite adeseori pentru a implementa sau extinde interfeţe sau clase (abstracte) specifice, clasa care le găzduieşte servind ca un furnizor de referinţe pentru a obţine implementări private ale respectivelor tipuri. Dacă o clasă internă nu este referită ca un tip specific, instanţele acesteia fiind tratate prin intermediul tipului implementat sau extins, se preferă definirea ei ca anonimă.

În Java există o sintaxă care permite derivarea unei clase obişnuite sau implementarea unei interfeţe printr-o clasă internă care nu este definită explicit sub un nume:

```
public class clasa_exterioară {
     clasa_părinte metodă_clasă_exterioară(){
       return new clasa_părinte(){
          // Urmează definiţia clasei interne anonime
          // - fără specificarea explicită a vreunui nume
             {<redefinire metode moştenite, definire
              noi membri sau implementare metode interfeţe>
             }
       }; sfârşit definiţie clasă internă anonimă
     } // încheiere definiţie metodă clasă exterioară
} // sfârşit definiţie clasă exterioară
```

Iniţializarea unei clase anonime se poate face prin constructorul implicit (invocat în lipsa definiţiei unui constructor explicit), însă poate se poate crea în mod explicit un constructor ca fiind o metodă fără însă a specifica şi un nume pentru aceasta.

```
public class clasa_exterioară {
     clasa_derivată metodă_clasă_exterioară(){
       return new clasa_derivată(){
          // Urmează definiţia clasei interne anonime
          // - fără specificarea explicită a vreunui nume
            {<redefinire metode moştenite, definire
            noi membri sau implementare metode interfeţe>
            // definire metodă constructor
            // fără un nume propriu-zis
                  {<instrucţiuni de iniţializare>}
```

```
            }
        }; sfârşit definiţie clasă internă anonimă
    } // încheiere definiţie metodă clasă exterioară
} // sfârşit definiţie clasă exterioară
```

Ca exemplu:

```
abstract class ClasaAbstracta1{
  private String m1 = "Valoare membru clasa interna";
  public abstract String getm1();
}
class ClasaExterna3 {
  public ClasaAbstracta1 referintaInterna(){
    return new ClasaAbstracta1(){
      private String m1 = "Valoare membru clasa interna anonima";
      // metoda de acces la membrul clasei interne
      public String getm1() {return m1;}
      {m1 = "Valoare membru clasa interna anonima preluata din constructor";}
    };
  }
}

public class TestInnerAnonim1 {
  public static void main(String[] args) {
    ClasaExterna3 obj_ext = new ClasaExterna3();
    ClasaAbstracta1 obj_abs = obj_ext.referintaInterna();
    System.out.println(obj_abs.getm1());
  }
}
```

Sintaxa folosită pentru definirea clasei interne de mai sus este de fapt o prescurtare pentru o secvenţă de cod cum ar fi:

```
class XClasaInterna extends ClasaAbstracta1 {
      private String m1 = "Valoare membru clasa interna anonima";
      // metoda de acces la membrul clasei interne
      public String getm1() {return m1;}
      public XClasaInterna2(){
          m1 = "Valoare membru clasa interna anonima preluata din constructor";
      }
}
```

Dacă în definiţia unei clase interne anonime este invocat (eventual preluat dintr-un argument de la un nivel superior) un obiect definit în afara acesteia, compilatorul va cere ca respectivul obiect extern să fie declarat *final*.

Clasele interne se formează şi se compilează ca oricare clase obişnuite, prin urmare vor avea propriile lor fişiere *.class* (fişierele conţinând codul compilat). Numele acestor fişiere (identificarea claselor interne) se formează printr-un şablon de genul:

Clasa_exterioară$Clasa_internă.class

sau generalizând

Clasa_ext$Clasa_ext_internă$...$Clasa_ internă.class.

Pentru clasele interne anonime acestea vor fi indicate prin nume în felul următor *Clasa_exterioară$1.class*, *Clasa_exterioară$2.class* etc.

2.3 Tratarea excepțiilor în Java

Tratarea excepțiilor (*exception handling*) reprezintă modalitatea prin care se pot soluționa problemele generate de erorile care pot apărea în momentul execuției, în scopul preîntâmpinării eşuării „fatale” a aplicațiilor la *runtime*.

Pe scurt, mecanismul de tratare a excepțiilor în Java se bazează pe o structură de instrucțiuni introdusă prin cuvântul cheie **try,** structură care găzduieşte codul prezumtiv a genera erori la *runtime*. Execuția instrucțiunilor incluse într-un astfel de bloc se va întrerupe la apariția unei erori, moment în care este generat un obiect-mesaj de tip *Exception* ce va putea fi preluat şi tratat într-o structură distinctă introdusă prin cuvântul cheie **catch**.

2.3.1 Genealogia excepțiilor: clasele fundamentale implicate

Excepțiile sunt încapsulate în obiecte ale căror tipuri derivă la bază din clasa **Throwable**. Erorile care nu pot fi captate şi tratate printr-un bloc *catch* derivă din clasa numită **Exception**. Prin urmare *throwable* semnifică un obiect care poate fi „aruncat” la execuție dintr-un anumit sector din aplicație şi gestionat în alt sector.

```
java.lang.Object
| ___ java.lang.Throwable
        | ___ java.lang.Exception
```

Clasa **Exception** derivă din clasa **Throwable** şi este folosită de fapt pentru a construi obiectele desemnând erorile de execuție. Metodele cele mai importante moştenite din clasa **Throwable** sunt următoarele:

- metoda prin care se poate obține un obiect *String* conținând textul mesajului de eroare;

```
String getMessage();
```

- metoda prin care se poate afişa direct *stream*-ul standard (întreaga stivă sau secvență de erori) generat în urma erorii de execuție;

```
void printStackTrace()
```

Alte clase care desemnează excepții şi care se pot dovedi utile sunt prezentate în figura următoare:

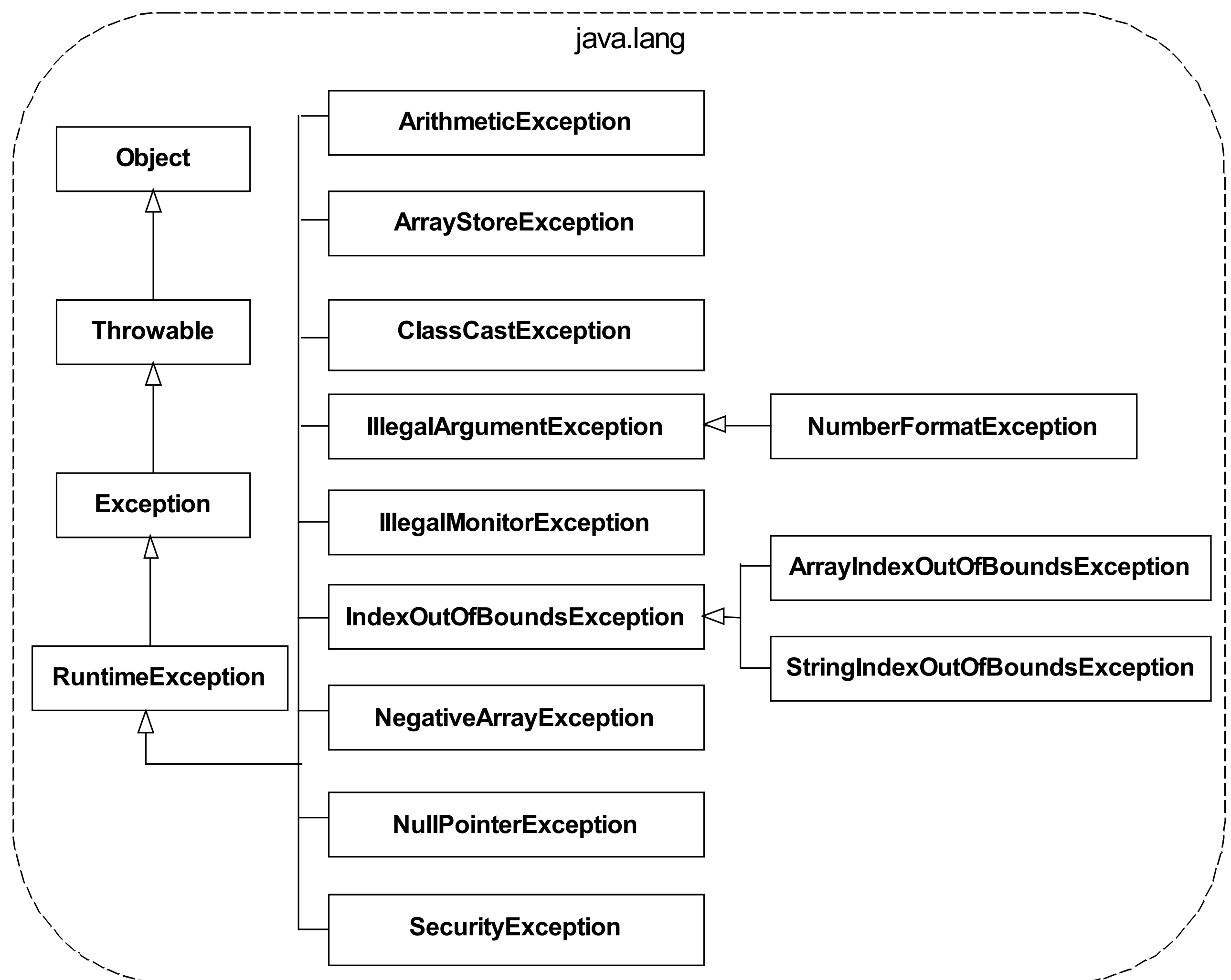


Figura 2-13 O parte din ierarhia de excepţii prezentă în mediul Java

Prin urmare excepţiile care interesează în mod deosebit sunt cele derivate din *RuntimeException* fiindcă acestea nu pot fi analizate şi verificate de către compilator. Tabelele următoare analizează excepţiile runtime cele mai importante:

Tabelul 2.1 Cele mai importante excepții ce pot apare la execuție

Clasa excepției	Descriere
ArtithmeticException	Cel mai des se referă la diviziunea cu zero
ArrayIndexOutOfBoundsException	Indexul indicat pentru un element dintr-un masiv (array) este mai mic ca zero sau mai mare sau cel puțin egală cu dimensiunea masivului
FileNotFoundException	Referința la un fişier nu poate fi soluționată
IllegalArgumentException	Apelul unei metode folosind un argument nepotrivit
IndexOutOfBoundsException	Indexul unui element/caracter dintr-un String sau Array nu este între limitele „legale”
NullPointerException	Indexul unui element/caracter dintr-un String sau Array nu este între limitele „legale”
NumberFormatException	Folosirea incorectă a unui format numeric, de obicei în apelul unei metode
StringIndexOutOfBoundsException	A index al unui caracter dintr-un *String* este mai mare ca zero sau este mai mare sau cel puțin egală cu dimensiunea String-ului

După cum am amintit mai înainte, atunci când din codul aflat în interiorul unui bloc **try** se generează o eroare de execuție aceasta poate fi tratată în blocul **catch(Exception exception)** care îi urmează. După ce codul **try/catch** este încheiat, apoi, indiferent dacă s-a generat sau nu o excepție sau dacă acea excepție a fost sau nu tratată într-o secțiune **catch**, va fi executat codul conținut într-o secțiune/bloc **finally**, bineînțeles dacă această secțiune există.

Iată un exemplu simplu de tratare a excepțiilor:

```
public class Excpt {
  public static void main(String[] args) {
    try{
      int[] tablou = new int[100];
      tablou[100] = 100;
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Eroare la executie: index invalid pentru un element dintr-un masiv !");
      e.printStackTrace();
    }
  }
}
```

De asemenea, există posibilitatea ca anumite proceduri/metode să poată retransmite, blocului sau metodelor din care au fost apelate, erori pentru a fi tratate la acel nivel. Cuvântul cheie pentru a realiza acest lucru este **throws**. Iată un exemplu în acest sens:

```
public class ExcptThrows {
  static void creareTablou() throws Exception{
    int[] tablou = new int[100];
    tablou[100] = 100;
  }
  public static void main(String[] args) {
    try {
      creareTablou();
    }
    catch(Exception e){
      System.out.println("Eroare la executie " + e.getLocalizedMessage());
      e.printStackTrace();

    }
  }
}
```

Blocurile **try** se pot include alte blocuri de acest gen. Implicațiile care decurg din acest fapt sunt următoarele: dacă unul din blocurile **try** nu are un bloc **catch** corespunzător care să trateze excepțiile atunci mediul (runtime) Java caută următorul bloc exterior pentru o secțiune **catch** căreia să-i transmită eventualele excepții. Iată un exemplu:

```
public class ExcptNestingTry {
  public static void main(String[] args) {
    try{
      try{
        String [] tablou = {"a", "b", "c", "d"};
        tablou[5] = "e";
      }
      catch(ArithmeticException e){
        System.out.println("Eroare aritmetica");
      }
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Eroare la executie ");
      e.printStackTrace();
    }
  }
}
```

2.3.2 Crearea propriilor excepții

Pe lângă faptul că un anumit bloc de instrucțiuni sau o anumită metodă poate retransmite eventualele erori de execuție pentru a fi tratate în contextul din care sunt invocate, există şi posibilitatea generării explicite a unei erori la un anumit punct sau moment dat. Instrucțiunea cheie pentru a realiza acest lucru este **throw**. De exemplu, următoarea secvență de cod este corectă:

```
public class TestThrow {
  public static void main(String[] args) {
    try {
      System.out.println("Inainte de generarea unei exceptii");
      throw new Exception();
    }
    catch (Exception e) {
```

```
      System.out.println("Din contextul tratarii exceptiei generate explicit");
    }
    System.out.println("Dupa generarea unei exceptii");
  }
}
```

Iar rezultatul va fi:

```
Inainte de generarea unei exceptii
Din contextul tratarii exceptiei generate explicit
Dupa generarea unei exceptii
```

Sintaxa generală pentru instrucţiunea **throw** este următoarea**:**

```
throw new ExceptionClassName(String MesajOpţional);
```

Iar sintaxa generală pentru **catch** este următoarea:

```
catch(ExceptionClassName numeParametru){
     // Instrucţiuni pentru tratarea excepţiei
}
```

Pentru specificarea faptului că o metodă ar putea genera anumite tipuri de erori care să fie tratate în contextul din care este apelată, sintaxa care ar trebui respectată este următoarea:

```
MetodeName throws ExceptionClassName1, ExceptionClassName1 ...{}
```

De asemenea, trebuie menţionat că există câteva restricţii faţă de instrucţiunile *try/catch/finally* şi anume:

- un bloc **try** trebuie urmat imediat de una (sau mai multe) blocuri **catch**, iar o instrucţiune **catch** trebuie obligatoriu să urmeze un bloc **try**. Iată un exemplu în care un bloc **try** este urmat de mai multe blocuri **catch**:

```
public class ExcptMultiCatch {
  public static void main(String[] args) {
    try {
      testExceptie(3);
    }
    catch(NumberFormatException e){
      System.out.println(e.getMessage());
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println(e.getMessage());
    }
    catch(NullPointerException e){
      System.out.println(e.getMessage());
    }
  }
  public static void testExceptie(int val) throws NumberFormatException,
   ArrayIndexOutOfBoundsException, NullPointerException{
    if (val == 1)
      throw new NumberFormatException("Exceptie pentru Numere");
    else if (val == 2)
      throw new ArrayIndexOutOfBoundsException("Exceptie pentru Array");
    else if (val == 3)
      throw new NullPointerException("Exceptie de referinta null-a");
    else
      System.out.println("OK");
  }
```

```
}
```

- o instrucțiune **throw** trebuie să presupună aria de acțiune a unui bloc **try**, iar tipul de excepție generat trebuie să se potrivească cu cel puțin una din clauzele **catch** ale acestuia.

Dincolo de aceste precizări, ar fi interesant ar fi dacă am putea genera şi trata *propriile excepții*, extinzând sistemul implicit de excepții al mediului *Java* cu excepții necesare contextului aplicației efective. Lucru este perfect realizabil cunoscând faptul că excepțiile sunt obiecte derivate în principal din clasa **Exception**, pot fi generate prin instrucțiunea **trow** şi pot fi tratate printr-un bloc **catch**. În acest sens poate fi realizată o excepție proprie poate fi generată ca în exemplul următor:

```
class ExceptieProprie extends Exception{
  int cod;
  ExceptieProprie(int pcod){
    cod = pcod;
  }
  public String toString(){
    return "Exceptie proprie " + cod;
  }
}

public class CustomException {
  public static void main(String[] args) {
    try {
      testExceptie(898989);
      testExceptie(2000);
    }
    catch(Exception e){
      System.out.println("Exceptie: " + e);
    }
    finally {
    System.out.println("Test incheiat");
    }
  }
  public static void testExceptie(int val) throws ExceptieProprie {
    if (val == 2000)
      throw new ExceptieProprie(val);
    else
      System.out.println("OK");
  }
}
```

Rezulatul final va fi:

```
OK
Exceptie: Exceptie proprie 2000
Test incheiat
```

2.4 Prelucrarea şirurilor

Şirurile sunt obiecte cu un mare grad de răspândire în mai toate aplicațiile scrise într-un limbaj orientat obiect, în speță Java. Ele reprezintă "materia primă" pentru complexele procesoare de text, dar şi pentru componente mai simple precum câmpurile, etichetele şi alte controale ale interfețelor grafice. De asemenea, modalitatea standard de afişarea a oricărui tip de obiect în Java se face cu ajutorul şirurilor, clasa *Object* prezentând o metodă specifică *toString()* apelată, de exemplu, prin *System.out.println()* şi care poate fi suprascrisă pentru a afişa informații mai „sugestive" decât echivalentul referinței de memorie al obiectului implicat.

Definiția cea mai comodă şi cea mai des întâlnită în prvința şirurilor invocă sintagma „colecție de date de tip caracter". Un *caracter* este un tip primitiv mai special, în sensul că reprezintă un element al unui sistem de codificare (ASCII) pentru simbolurilor folosite în lucrul cu computerul. Un caracter are echivalent, ca reprezentare în memoria internă, un cod *integer*. În Java o valoare *char* poate fi promovată (casting implicit) într-un *int*, obținând ca urmare codul ASCII corespunzător, şi, invers, un *int* poate face obiectul unei operații de casting explicite la un *char,* obținând caracterul corespunzător (dacă respectivul întreg există în pagina de cod). De exemplu, în urma execuției următorului test:

```
public class TestCaractere {
   public TestCaractere() {
   }
   public static void main(String[] args) {
      char caracter = 'z';
      System.out.println(caracter);
      System.out.println((int)caracter);
      int cod = 122;
      System.out.println(cod);
      System.out.println((char)cod);
   }
}
```

vom obține:

```
z
122
122
z
```

Un *şir de caractere* reprezintă însă o structură de date mult mai complexă, şi a fost implementată ca un obiect "complet" în Java prin clasa *String* care prezintă o structură destul de complexă impplicând metode legate de conversii, de căutare în interiorul şirurilor, de comparare a şirurilor etc.

2.4.1 Obținerea şirurilor de caractere

Clasa *String* din biblioteca standard a distribuției Java are, aşa cum se obişnuieşte, proprii constructori:

```
// obține un şir de caractere "empty"
public String()
// obține o copie a sirului specificat ca parametru
public String(String initial_value)
```

De cele mai multe ori obiectele tip *String* sunt specificate delimitând-ul prin caractere specifice, de exemplu:

```
String sir1 = "sir de caractere";
```

Prin urmare, "intrarea în scenă" a unui astfel de obiect se realizează odată cu specificarea lui în modul exemplificat mai sus. Astfel o nouă variabilă tip *String* inițializată printr-o expresie asemănătoare cu cea de mai sus

```
String sir2 = "sir de caractere";
```

nu va realiza nimic spectaculos, în sensul că nu va obține un nou obiect ci va conține o referință către acelaşi obiect-şir de caractere inițializat înainte. Ca urmare ultima instrucțiune este echivalentă cu:

```
String sir2 = sir1;
```

2.4.2 Conversia datelor în şiruri de caractere

Este binecunoscut că valoarea 111 şi "111" sunt complet diferite. În Java, această diferență este dată de faptul că prima reprezintă un primitiv numeric (fie el *byte, short, int, long, float* sau *double*), iar ce de a doua o instanță *String*. Cum însă s-ar putea obține programatic a doua valoarea având la bază prima valoare ? În acest sens, clasa *String* pune la dospoziție o suită de metode *valueOf()* pentru toate tipurile de primitive, cât şi pentru array-urile de caractere (char) sau chiar obiecte. Membrul *valueOf()* este o metodă statică, prin urmare poate fi folosită în felul următor:

```
String n = new String(String.valueOf(128));
String b = new String(String.valueOf(true));
String c = new String(String.valueOf('c'));
String ac = new String(String.valueOf(new char[]{'a', 'b', 'c'}));
```

2.4.3 Indexarea şirurilor: căutarea sau regăsirea caracterelor individuale şi a subşirurilor

Am aminit deja că şirurile reprezintă *colecții* de caractere (*char*) aşa cum sunt vectorii sau array-urile. Acest lucru are cel puțin două implicații:

- fiecare obiect *String* are o anumită lungime (proprietatea *length*) care reprezintă numărul de caractere ce formează şirul;
- fiecare element al şirului (fiecare caracter - *char*) este indexat funcție de poziția sa în colecție, pornind de la zero.

Ca urmare, clasei *String* i-a fost asociată o întreagă funcționalitate legată de căutare/poziționare/regăsire a caracterelor pe baza indecşilor elementelor din şir. Astfel:

- metodele *indexOf()* caută un caracter (sau chiar un şir de caractere) de la stânga la dreapta pornind de la prima poziție sau de la alta al cărei index este specificat explicit;
- metodele *lastIndexOf()*caută un caracter (sau chiar un şir de caractere) de la dreapta la stânga pornind de la ultima poziție sau de la alta al cărei index este specificat explicit.

```
public int indexOf(int caracter)
```

```
public int indexOf(int caracter, int pozițieStart)
public int indexOf(String şir)
public int indexOf(String şir, int pozițieStart)
public int lastIndexOf(int caracter)
public int lastIndexOf(int caracter, int pozițieStart)
public int lastIndexOf(String şir)
public int lastIndexOf(String şir, int pozițieStart)
```

Pentru a obține însă caracterul situat pe o anumită poziție într-un şir se foloseşte metoda *charAt()*, iar dacă se doreşte un subşir dintr-un şir indicându-i poziția primului (şi eventual ultimului) caracter se folosesc metodele *subString()*:

```
public char charAt(int index)
public String subString(int startIndex)
public String subString(int startIndex, int sfârşitIndex)
```

Astfel de metode ne-ar putea folosi la interpretarea datelor specificate sub forma unui şir de caractere, dar având componente distincte delimitate prin caractere specifice. Spre exemplu, şirul de conectare la o bază de date Oracle conține trei componente: nume, parolă şi serviciuBD. În acest sens am putea construi o clasă care să interpreteze acest şir şi să returneze fiecare componentă în parte:

```
public class OraConnect {
    private String sirConectare;
    private String utilizator;
    private String parola;
    private String serviciuBD;
    /** Creates a new instance of OraConnect */
    public OraConnect(String conect) {
        sirConectare = conect;
        parseSirConectare();
    }
    private void parseSirConectare(){
        //format sir conectare user/parola@serviciu
        int pozitieDel1 = sirConectare.indexOf('/');
        int pozitieDel2 = sirConectare.indexOf('@');
        utilizator = sirConectare.substring(0, pozitieDel1);
        parola = sirConectare.substring(pozitieDel1 + 1, pozitieDel2);
        serviciuBD = sirConectare.substring(pozitieDel2 + 1);
    }
    public String getUtilizator(){
        return utilizator;
    }
    public String getParola(){
        return parola;
    }
    public String getServiciu(){
        return serviciuBD;
    }

    public static void main(String[] args){
        OraConnect sir = new OraConnect("scott/tiger@BDSTUD");
        System.out.println("User: " + sir.getUtilizator());
        System.out.println("Password: " + sir.getParola());
        System.out.println("Serviciu: " + sir.getServiciu());
    }
}
```

În urma execuției vom obține:

```
User: scott
Password: tiger
Serviciu: BDSTUD
```

2.4.4 Compararea şirurilor de caractere

Egalitatea şi identitatea obiectelor în general par cam acelaşi lucru, şi totuşi nu sunt, sau, cel puțin, sunt necesare câteva nuanțări. De obicei, în codul sursă, obiectele sunt invocate prin „numele" lor, sau, mai bine zis, prin numele variabilelor care fac referire la ele. Când punem operatorul de egalitate (= =) între două variabile, atunci de fapt verificăm conținutul celor două variabile. Prin urmare dacă cele două variabile au acelaşi conținut pot fi considerate „egale". Însă, în programarea obiectuală, variabilele de tipuri ne-primitive conțin referințe la obiectele ca atare şi, în consecință, „egalitatea" semnifică în acest caz că cele două variabile fac referire la acelaşi obiect. Şi totuşi, de cele mai multe ori ceea ce se vizează prin „egalitate" se referă la „echivalența" obiectelor ca atare şi nu a referințelor variabilelor. În acest sens clasa obiect oferă metoda *equals()* care, deşi implicit verifică „egalitatea" referințelor, poate fi suprascrisă astfel încât variabile cu referințe diferite să fie considerate egale ca urmare a unui criteriu care să țină seama de intimitatea obiectelor şi nu de identitatea lor (referințele).

În legătură cu şirurile de caractere (instanțele *String*) metodele de comparare sunt următoarele:

```
public boolean equals(Object anObject)
public boolean equalsIgnoreCase(String anotherString)
public int compareTo(String anotherString)
```

Prima metodă *equals()* suprascrie metoda *Object.equals()* aşa încât „egalitatea" nu mai este probată prin intermediul referințelor, ci două obiecte *String* sunt considerate egale dacă au exact acelaşi set de caractere (bineînțeles în aceeaşi ordine). Cea de a două metodă *equalsIgnoreCase()* merge chiar mai departe şi, în momentul verificării, presupune echivalente minusculele şi majusculele ale aceluiaşi simbol literal, deşi reprezintă două caractere diferite. Metoda *compareTo()* returnează o valoare *int* care evaluează nu numai egalitatea (caz în care întoarce 0) ci şi dacă obiectul de comparație precedă obiectul inițial (caz în care întoarce o valoare pozitivă) sau îi urmează acestuia (caz în care întoarce o valoare negativă).

În sensul celor afirmate mai sus, iată următorul exemplu:

```
public class TestEgalitateSiruri {
   public static void main(String[] args) {
      String sir1 = "Primul";
      String sir2 = "Altul";
      String sir3 = sir2;
      String sir4 = "Altul";
      String sir5 = "primul";
      String sir6 = new String("Primul");

      System.out.println("Identitate: sir1==sir3 --> " + (sir1==sir3));
      System.out.println("Egalitate: sir1.equals(sir3) --> " + (sir1.equals(sir3)));
      System.out.println("Identitate: sir2==sir4 --> " + (sir2==sir4));
      System.out.println("Egalitate: sir2.equals(sir4) --> " + (sir2.equals(sir4)));
      System.out.println("Identitate: sir1==sir5 --> " + (sir1==sir5));
```

```
        System.out.println("Egalitate: sir1.equalsIgnoreCase(sir5) --> " +
    (sir1.equalsIgnoreCase(sir5)));
        System.out.println("Identitate: sir1==sir6 --> " + (sir1==sir6));
        System.out.println("Egalitate: sir1.equals(sir6) --> " + (sir1.equals(sir6)));
    }
}
```

care, la execuție, determină următorul rezultat:

```
Identitate: sir1==sir3 --> false
Egalitate: sir1.equals(sir3) --> false
Identitate: sir2==sir4 --> true
Egalitate: sir2.equals(sir4) --> true
Identitate: sir1==sir5 --> false
Egalitate: sir1.equalsIgnoreCase(sir5) --> true
Identitate: sir1==sir6 --> false
Egalitate: sir1.equals(sir6) --> true
```

2.4.5 Modificarea şi parcurgerea şirurilor. Clasele StringBuffer şi StringTokenizer

Şirurile au totuşi un mare neajuns, sunt *imuabile*, sau, cu alte cuvinte, odată instanțiate nu pot fi modificate în nici un fel. Folosirea operatorului de concatenare duce implicit la crearea unor noi obiecte de tip *String*, iar o astfel de operație într-o structură iterativă se dovedeşte extrem de neperformantă şi în ceea ce priveşte consumul de resurse de memorie şi în ceea ce priveşte viteza de execuție. Pentru a depăşi această problema distribuția Java ne aduce în ajutor o clasă specială, nederivată din *String*, dar care lucrează tot cu şiruri de caractere, clasa *StringBuffer*.

Un *StringBuffer* se recomandă a fi utilizat în locul unui *String* pentru orice fel de operații care implică şiruri de caractere. Această clasă asigură prin constructorului ei *StringBuffer(String)* şi prin metoda *toString()* o cale simplă şi directă de conversie a şirurilor de caractere în instanțe *StringBuffer* şi invers. Principala virtute a acestei clase este că *şirurile de caractere* pe care le conțin instanțele sale pot fi modificate prin metode gen *insert(int poziție, String data)* sau *append(String data)*.

Pentru a arăta modul în care clasa *StringBuffer* poate fi folosită în locul clasei *String,* în anumite cazuri, să luăm în calcul o ipoteză de lucru, inversă față de exemplul prezentat în paragraful despre indexarea şirurilor de caractere, şi anume să construim şirul de conectare la o bază de date pornind de la numele utilizatorului, parola acestuia şi numele serviciului. Varianta, să-i zicem „clasică”, ar fi următoarea:

```
public class OraConnectString {
    private String sirConectare;
    private String utilizator;
    private String parola;
    private String serviciuBD;
    public OraConnectString (String user, String password, String service) {
        utilizator = user;
        parola = password;
        serviciuBD = service;
        makeStringConnect();
    }
    private void makeStringConnect(){
        sirConectare = utilizator + '/' + parola + '@' + serviciuBD;
```

```
    }
    public String getSirConectare(){
        return sirConectare;
    }
    public static void main(String[] args) {
        System.out.println(new OraConnectString_1("scott", "tiger",
  "BDSTUD").getSirConectare());
    }
}
```

Folosind clasa *StringBuffer*, putem rescrie metoda *makeStringConnect* astfel:

```
public class OraConnectString {
    private String sirConectare;
    private String utilizator;
    private String parola;
    private String serviciuBD;
    public OraConnectString (String user, String password, String service) {
        utilizator = user;
        parola = password;
        serviciuBD = service;
        makeStringConnect();
    }
    private void makeStringConnect(){
        StringBuffer sir = new StringBuffer();
        sir.append(utilizator);
        sir.append('/');
        sir.append(parola);
        sir.append('@');
        sir.append(serviciuBD);
        sirConectare = sir.toString();
    }
    public String getSirConectare(){
        return sirConectare;
    }
    public static void main(String[] args) {
        System.out.println(new OraConnectString_1("scott", "tiger",
  "BDSTUD").getSirConectare());
    }
}
```

Exemplul de mai sus relevă, într-o oarecare măsură modul de utilizare al clasei *StringBuffer*, însă nu prea arată utilitatea acesteia.

Să luăm în considerare o altă problemă care ne propune reconstituirea unui şir de caractere din elementele individuale ale unui tablou(array). În acest scop, am putea construi o clasă după cum urmează:

```
public class StringList {
    private String[] listItems;
    private String listString;
    private StringBuffer listBuffer = new StringBuffer();
    private String delimitator = ",";
    /** Creates a new instance of StringList */
    public StringList(String[] list) {
        listItems = new String[list.length];
        for(int i = 0; i < list.length; i++){
            listItems[i] = list[i];
            if (i==0)
                listString += list[i];
            else
```

```
            listString += delimitator + list[i];
        }
    }
    public String getList(){
        return listString;
    }
}
```

Constructorul clasei *StringList* reconstituie din elementele argumentului său de tip *array* şirul de caractere care va fi accesebil prin metoda *getList()*. Neajunsul algoritmului propus este faptului că la fiecare iterație în structura de control a parcurgerii tabloului *list* este creat un nou şir de caractere (o nouă instanță *String*) a cărui referință este preluată în membrul *listString*. Prin urmare la fiecare pas va fi creat un nou obiect, ceea ce din punctul de vedere al gestiunii memoriei nu este o soluție foarte fericită fiindcă instanțirea unei clase este o operație consumatoare de resurse (atât în ceea ce priveşte timpul de procesare, cât şi în ceea ce priveşte memoria alocată). Folosind însă clasa *StringBuffer* putem optimiza operația de reconstituire a şirului astfel:

```
public class StringList {
    private String[] listItems;
    private String listString;
    private StringBuffer listBuffer = new StringBuffer();
    private String delimitator = ",";
    /** Creates a new instance of StringList */
    public StringList(String[] list) {
        listItems = new String[list.length];
        for(int i = 0; i < list.length; i++){
            if (i==0)
                listBuffer.append(list[i]);
            else
                listBuffer.append(delimitator + list[i]);
        }
        listString = listBuffer.toString();
    }
    public String getList(){
        return listString;
    }
}
```

Deosebirea rezidă în faptul că nu mai este creat la fiecare pas un nou obiect *String*, ci, odată creată instanța *StringBuffer*, aceasta este completată la fiecare iterație cu elementul corespunzător de pe poziția *i* a tabloului *list*.

Dacă însă punem „pe tapet" şi problema inversă: dintr-un şir de caractere să extragem elementele individuale despărțite printr-un delimitator ? În acest caz, clasa care ne poate oferi sprijin este *StringTokenizer*. Prin urmare să adăugăm o nouă metodă (constructor) în clasa *StringList* care să preia ca argumente o listă sub forma unui şir şi să o redea sub forma unui *array* ale cărui elemente să fie obiecte de tip *String*:

```
    public StringList(String list){
        StringTokenizer stoke = new StringTokenizer(list, delimitator);
        listItems = new String[stoke.countTokens()];
        listString = list;
        String item;
```

```
        int i = 0;
        while (stoke.hasMoreElements()){
            item = stoke.nextToken();
            listItems[i] = item;
            i++;
        }
    }
    public String[] getListItems(){
        return listItems;
    }
```

Algoritmul de lucru este următorul: pentru lista-şir de caractere se creează o instanţă *StringTokenizer*. Rolul acestei clase este să „analizeze" şirul funcţie de un *delimitator* specific (vezi valoarea acestui membru în definiţia clasei *StringList*). Utilitatea *StringTokenizer* este evidentă datorită metodei *nextToken()*, care returnează următorul element delimitat după cum s-a stabilit la instanţierea clasei. De asemenea, metoda *countTokens()* returnează numărul de fragmente ale şirului iniţial, iar metoda *hasMoreElements()* este utilizată împreună cu *nextToken()* pentru a forma o structură bine formalizată de parcurgere a şirului iniţial.

Prin urmare, listingul complet al clasei *StringList* ar putea fi următorul (inclusiv o secvenţă *main* de test):

```
import java.util.StringTokenizer;
public class StringList {
    private String[] listItems;
    private String listString;
    private StringBuffer listBuffer = new StringBuffer();
    private String delimitator = ",";
    /** Creates a new instance of StringList */
    public StringList(String[] list) {
        listItems = new String[list.length];
        for(int i = 0; i < list.length; i++){
            listItems[i] = list[i];
            //listString += delimitator + list[i];
            if (i==0)
                listBuffer.append(list[i]);
            else
                listBuffer.append(delimitator + list[i]);
        }
        listString = listBuffer.toString();
    }
    public StringList(String list){
        StringTokenizer stoke = new StringTokenizer(list, delimitator);
        listItems = new String[stoke.countTokens()];
        listString = list;
        String item;
        int i = 0;
        while (stoke.hasMoreElements()){
            item = stoke.nextToken();
            listItems[i] = item;
            i++;
        }
    }
    public String getList(){
        return listString;
    }
    public String[] getListItems(){
        return listItems;
    }
    public static void main(String[] args) {
```

```
        StringList sList = new StringList(new String[]
            {"primul",
             "al doilea",
             "al treilea",
             "al patrulea",
             "al cincilea"});
        System.out.println(sList.getList());

        sList = new StringList("primul,al doilea,al treilea,al patrulea,al cincilea");
        for(int i=0; i < sList.getListItems().length; i++){
            System.out.println(sList.getListItems()[i]);
        }
    }
}
```

În urma execuției vom obține următorul rezultat:

```
primul,al doilea,al treilea,al patrulea,al cincilea
primul
al doilea
al treilea
al patrulea
al cincilea
```

Arhitecții limbajului Java au considerat însă clasa *StringTokenizer* totuşi destul de anevoios de manipulat, astfel încât pentru „fragmentarea” unui şir folosind un delimitator, au îmbogățit în JDK 1.4 clasa *String* cu metoda

`public String[] split(String delimitator)`

Astfel încât laborioasa metodă de parcurgere a şirului cu *hasMoreElements()* şi *nextToken()* poate fi înlocuită pur şi simplu cu expresia

```
listItems = list.split(delimitator);
```

2.5 Egalitatea şi comparabilitatea obiectelor

2.5.1 Egalitate vs. identitate

Când am discutat despre modul de comparare a *String*-urilor am amintit despre conceptele de *egalitate* şi *identitate*. A sosit momentul să generalizăm aceste principii la nivelul oricărui tip de obiecte, ceea ce vom face în continuare.

Referențierea obiectelor este principiul care face distincția între *egalitate* şi *identitate*: dacă două „viziuni” distincte fac referire la acelaşi obiect (aceeaşi locație de memorie în care se găseşte stocat un obiect) atunci vorbim despre *identitate*, iar dacă două „viziuni” distincte fac referire la două obiecte diferite, atunci putem vorbi cel mult de „echivalență” (să nu-i spunem egalitate, încă). În acest sens figura următoare este, sperăm, edificatoare.

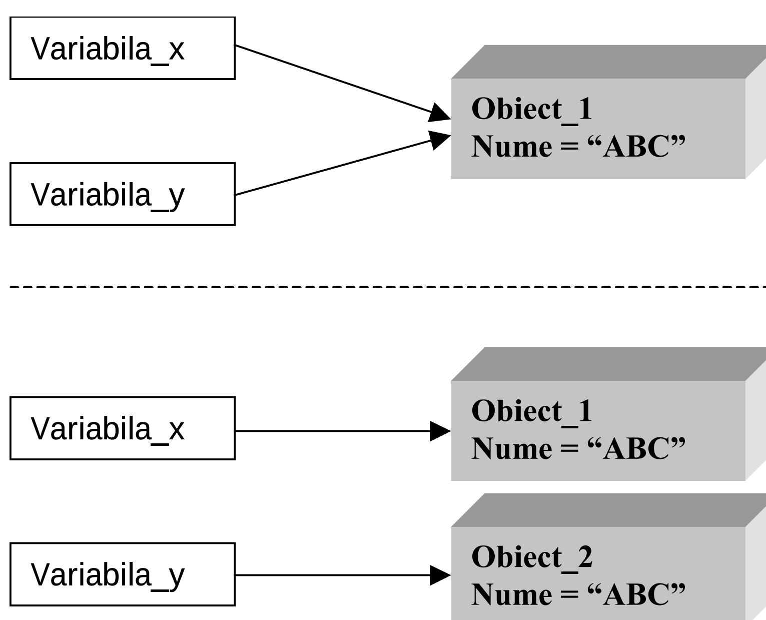


Figura 14 Identitatea vs. egalitate

Astfel, în prima situație vorbim despre identitate, iar în a doua situație vorbim despre „echivalență” dacă vom lua în cosiderare proprietatea *nume* a celor două obiecte.

De multe ori (sau poate de cele mai multe ori), când se compară două obiecte se are în vedere „echivalența” lor sub anumite aspecte şi nu identitatea acestora. În programarea orientată obiect, în speță în Java, problema a fost rezolvată prin intermediul *suprascrierii* (polimorfice) a metodei *equals()* aparținând clasei *Object* care reprezintă rădăcina ierarhiei de moştenire. Astfel, metoda *equals()* în mod implicit vizează identitatea obiectelor la fel ca în cazul operatorului relațional de egalitate „= =” (de fapt, interpretarea acestui operator se realizează prin intermediul metodei *equals()* de la nivelul clasei *Object*). Prin suprascriere însă, această metodă de comparare poate fi redefinită la nivelul subclaselor, „egalitatea” putând fi testată funcție de alte criterii care să nu implice identitatea (funcția operatorului “= =” nu va putea fi însă suprascrisă).

Pentru relevanță putem lua în considerare exemplul următor în care pentru clasa *Persoana* am redefinit metoda *equals* luînd în considerare valoarea membrului *cnp:*

```
class Persoana{
   public String cnp;
```

```
    public String nume;
    public String prenume;
    public Persoana(String pCnp, String pNume, String pPrenume){
        cnp = pCnp;
        nume = pNume;
        prenume = pPrenume;
    }
    public boolean equals(Object obj){
        if (obj instanceof Persoana)
            return (cnp == ((Persoana)obj).cnp);
        else
            return false;
    }
}

public class TestEgalitate {
    public static void main(String[] args) {
        Persoana p1 = new Persoana("CNP001", "Ion", "Viorel");
        Persoana p2 = new Persoana("CNP001", "Ion", "Viorel");
        System.out.println("p1==p2 --> " + (p1 == p2));
        System.out.println("p1.equals(p2) --> " + p1.equals(p2));
    }
}
```

Se remarcă faptul că am făcut mai întâi un test prin care am testat dacă obiectul comparat este de tipul *Persoana*, metoda redefinită *equals()* dorind să fie valabilă numai pentru instanțe din aceeaşi clasă.

La execuție rezultatul va fi următorul:

```
p1==p2 --> false
p1.equals(p2) --> true
```

2.5.2 Paradoxuri ale testului de egalitate

Există situații mai speciale în care pot apare anumite fenomene aparent paradoxale. Să extidem exemplul anterior derivând din *Persoana* clasa *Student*, care are la rândul ei o metodă *equals()*:

```
class Persoana{
    public String cnp;
    public String nume;
    public String prenume;
    public Persoana(String pCnp, String pNume, String pPrenume){
        cnp = pCnp;
        nume = pNume;
        prenume = pPrenume;
    }
    public boolean equals(Object obj){
        if (obj instanceof Persoana)
            return (cnp == ((Persoana)obj).cnp);
        else
            return false;
    }
}
class Student extends Persoana{
    public String matricol;
    public String specializare;

    public Student(String pCnp, String pMatricol, String pNume,
```

```
      String pPrenume, String pSpecializare){
         super(pCnp, pNume, pPrenume);
         matricol = pMatricol;
         specializare = pSpecializare;
      }
      public boolean equals(Student stud){
         return (matricol == stud.matricol);
    }
}
public class TestEgalitate {
   public static void main(String[] args) {
      Student s1 = new Student("CNP001", "MT001", "Ion", "Viorel", "Spec1");
      Student s2 = new Student("CNP001", "MT002", "Ion", "Viorel", "Spec2");

      System.out.println("s1.equals((Persoana)s2) -->" + s1.equals((Persoana)s2));
      System.out.println("s1.equals(s2) --> " + s1.equals(s2));
   }
}
```

Ca urmare a execuției testului vom obține următorul rezultat:

```
s1.equals((Persoana)s2) -->true
s1.equals(s2) --> false
```

Prin urmare la nivelul „părintelui” instanțele clasei „copil” au fost considerate egale, lucru evident din primul test unde am făcut o operație de casting asupra argumentului metodei *equals()* pentru a determina legarea ei la nivelul clasei *Persoana*. Pentru a redefini modalitatea de comparare la nivelul clasei *Student* am supraîncărcat (atenție! Nu am suprascris) metoda *equals()* cu o definiție care să țină seama de valoare membrului *matricol*.

2.5.3 Compararea obiectelor pentru operații de sortare a array-urilor: interfețele *Comparable* şi *Comparator*

Compararea obiectelor nu se face în totdeauna în sensul de „egalitate”, există şi expresii care implică sintagme gen ”mai mare” sau „mai mic”. Acest gen de operații sunt întâlnite cel mai des asupra datelor (primitive) numerice, însă ele se dovedesc utile şi pentru operațiile de sortare a colecțiilor de obiecte.

În acest sens, biblioteca distribuției Java ne oferă două interfețe:

- `java.lang.Comparable` – clasele care implementează această interfață vor defini o metodă prin care pot fi ordonate instanțele acestora:

  ```
  public int compareTo(java.lang.object obj)
  ```

  Metoda *compareTo()* va compara obiectul pentru care este invocată cu argumentul acesteia şi va returna un întreg (*int*) negativ dacă este acesta este considerat “inferior”. În caz de egalitate va returna zero (0) şi în cazul în care obiectul este considerat “superior” argumentului va returna un întreg pozitiv. Clasa *String* implementează spre exemplu această interfață pentru a asigura ordonarea alfabetică a şirurilor de caractere.
- `java.util.Comparator` – clasele care implementează această interfață furnizează un serviciu de ordonare specific pentru instanțele altor clase. Metodele impuse de această interfață sunt următoarele:

  ```
  public int compare(Object ob_stg, Object ob_drpt);
  ```

```
public boolean equals(Object comp);
```

Metoda *compare()* returnează un întreg (*int*) negativ, zero sau pozitiv pentru < (mai mic), egal sau > (mai mare) în acelaşi mod ca şi metoda *compareTo()* din interfaţa *Comparable*.

Interfaţa *Comparable* este implementată de toate clasele care „împachetează" primitive, cum sunt *Integer, Double, Float, Long* etc.

Aceste două interfeţe se dovedesc foarte utile în sortarea colecţiilor de obiecte. Pentru a exemplifica să luăm cazul sortării elementelor unui tablou *array* de *Persoane*. În acest scop preluăm clasa *Persoana* din exemplele anterioare şi o modificăm astfel:

```
class Persoana implements Comparable{
   public String cnp;
   public String nume;
   public String prenume;
   public Persoana(String pCnp, String pNume, String pPrenume){
      cnp = pCnp;
      nume = pNume;
      prenume = pPrenume;
   }
   public boolean equals(Object obj){
      if (obj instanceof Persoana)
         return (cnp == ((Persoana)obj).cnp);
      else
         return false;
   }

   public int compareTo(Object o) {
      Persoana c = (Persoana) o;
      if (nume.equals(c.nume))
         return prenume.compareTo(c.prenume);
      else
         return nume.compareTo(c.nume);
   }
   public String toString(){
      return cnp + " " + nume + " " + prenume;
   }
}
```

După cum se observă clasa *Persoana* a implementat interfaţa *Comparable*, iar în metoda *compareTo()* ordonarea instantelor „persoane" se face în primul rând după nume, şi apoi după prenume.

Pentru a exemplifica modul de ordonare, am definit o clasă de test care în metoda sa *sort,* parametrizată cu un array de elemente *Comparable,* implementează algoritmul „select-sort". Iată definiţia acestei clase:

```
public class TestSort {
   public static void main(String[] args) {
      Comparable[] lista = new Persoana[]{
        new Persoana("CNP2", "Popescu", "Marin"),
        new Persoana("CNP4", "Ion", "Viorel"),
        new Persoana("CNP5", "Cimpeanu", "Gheorghe"),
        new Persoana("CNP3", "Stoica", "Viorel"),
        new Persoana("CNP1", "Ion", "Vasile"),
      };
      sort(lista);
      //System.out.println(lista[1].compareTo(lista[4]));
```

```
    }
    public static void sort(Comparable[] lst){
        // aplic algoritmul Select-Sort
        int idx;
        Comparable temp;
        for(int i=0; i<lst.length; i++){
            idx = i;
            for(int j=i+1; j<lst.length; j++){
                if (lst[j].compareTo(lst[idx])<0){
                    idx = j;
                }
            }
            temp = lst[i];
            lst[i] = lst[idx];
            lst[idx] = temp;
            System.out.println("P" + i);
            printList(lst);
        }
        // afisez lista sortata
        System.out.println("Final");
        printList(lst);
    }
    public static void printList(Object[] lst){
        for(int i=0; i < lst.length; i++)
            System.out.println(lst[i]);
    }
}
```

Rezultatul ar trebui să fie (paşii intermediari şi final) următorul:

```
P0
CNP5 Cimpeanu Gheorghe
CNP4 Ion Viorel
CNP2 Popescu Marin
CNP3 Stoica Viorel
CNP1 Ion Vasile
P1
CNP5 Cimpeanu Gheorghe
CNP1 Ion Vasile
CNP2 Popescu Marin
CNP3 Stoica Viorel
CNP4 Ion Viorel
P2
CNP5 Cimpeanu Gheorghe
CNP1 Ion Vasile
CNP4 Ion Viorel
CNP3 Stoica Viorel
CNP2 Popescu Marin
P3
CNP5 Cimpeanu Gheorghe
CNP1 Ion Vasile
CNP4 Ion Viorel
CNP2 Popescu Marin
CNP3 Stoica Viorel
P4
CNP5 Cimpeanu Gheorghe
CNP1 Ion Vasile
CNP4 Ion Viorel
CNP2 Popescu Marin
CNP3 Stoica Viorel
Final
CNP5 Cimpeanu Gheorghe
CNP1 Ion Vasile
CNP4 Ion Viorel
```

```
CNP2 Popescu Marin
CNP3 Stoica Viorel
```

Interfaţa *Comparator* ne-ar putea ajuta să definim mai multe criterii de ordonare astfel implementate:

```
class ComparatorCNP implements java.util.Comparator{
   public int compare(Object o1, Object o2) {
      Persoana p1 = (Persoana) o1;
      Persoana p2 = (Persoana) o2;
      return p1.cnp.compareTo(p2.cnp);
   }
   public String toString(){
      return "dupa CNP";
   }
}

class ComparatorNumePren implements java.util.Comparator{
   public int compare(Object o1, Object o2) {
      Persoana p1 = (Persoana) o1;
      Persoana p2 = (Persoana) o2;
      if (p1.nume.equals(p2.nume))
         return p1.prenume.compareTo(p2.prenume);
      else
         return p1.nume.compareTo(p2.nume);
   }
   public String toString(){
      return "dupa nume si prenume";
   }
}
```

Numele celor două clase este sugestiv, aşa că nu mai merită insistat asupra funcţionalităţii lor. Pentru a le valorifica, modificăm clasa *TestSort* astfel:

```
public class TestSort {
   public static void main(String[] args) {

   public static void main(String[] args) {
      java.util.Comparator comparatorCNP = new ComparatorCNP();
      java.util.Comparator comparatorNumePren = new ComparatorNumePren();
      Comparable[] lista = new Persoana[]{
        new Persoana("CNP2", "Popescu", "Marin"),
        new Persoana("CNP4", "Ion", "Viorel"),
        new Persoana("CNP5", "Cimpeanu", "Gheorghe"),
        new Persoana("CNP3", "Stoica", "Viorel"),
        new Persoana("CNP1", "Ion", "Vasile"),
      };
      sort(lista, comparatorNumePren);
      sort(lista, comparatorCNP);
   }
   public static void sort(Comparable[] lst, java.util.Comparator comparator){
      // aplic algoritmul Select-Sort
      int idx;
      Comparable temp;
      for(int i=0; i<lst.length; i++){
         idx = i;
         for(int j=i+1; j<lst.length; j++){
            if (comparator.compare(lst[j],lst[idx])<0){
               idx = j;
            }
         }
         temp = lst[i];
         lst[i] = lst[idx];
         lst[idx] = temp;
```

```
        }
        // afisez lista sortata
        System.out.println(comparator);
        printList(lst);
    }
    public static void printList(Object[] lst){
        for(int i=0; i < lst.length; i++)
            System.out.println(lst[i]);
    }
}
```

Prin urmare, metoda *sort* a fost modificată astfel încât criteriul de sortare să fie parametrizat folosind un obiect de tip *Comparator*, în algoritmul de sortare intervenind următoare modificare:

în loc de

```
            if (lst[j].compareTo(lst[idx])<0){
```

am folosit

```
            if (comparator.compare(lst[j],lst[idx])<0){
```

iar rezultatul final va fi următorul:

```
dupa nume si prenume
CNP5 Cimpeanu Gheorghe
CNP1 Ion Vasile
CNP4 Ion Viorel
CNP2 Popescu Marin
CNP3 Stoica Viorel
dupa CNP
CNP1 Ion Vasile
CNP2 Popescu Marin
CNP3 Stoica Viorel
CNP4 Ion Viorel
CNP5 Cimpeanu Gheorghe
```

2.6 Colecții de obiecte

Colecțiile utilizate cel mai adesea pentru păstrarea grupurilor de obiecte sunt *array*-urile obținute (deşi prin sintaxă nu se face explicit vreo referire) folosind clasa *Array*. Acest tip de colecții are însă câteva limite, dintre care, poate cea mai „supărătoare", este dificultatea redimensionării. De asemenea, accesul unui element dintr-un *array* se poate face numai prin indexul său. Platforma Java pune, din fericire, la dispoziția programatorilor o întreagă „serie" de clase ce dau forme variate *colecțiilor* de obiecte.

2.6.1.1 Imagine generală asupra bibliotecii de clase privind colecțiile

Pentru a da un sens mai larg termenului „colecții" unii autori folosesc noțiunea de *container* în sensul de întreg format din mai multe „părți" omogene sau neomogene în ceea ce priveşte tipurile lor.

Prima distincție între aceste clase se face din *punctul de vedere al accesului* şi anume:

- pe de o parte avem colecții în care accesul la obiecte se face prin index (asemănător *array*-urilor). Din punctul de vedere al utilizatorului aceste colecții sunt „văzute" ca fiind de tip *Collection*, care este de fapt o interfață;
- pe de altă parte avem colecții în care obiectelor le sunt asociate chei, o cheie putând face legătura cu un singur obiect. Evident accesul se bazează pe aceste chei care nu sunt obligatorii de un anumit tip (primitiv sau nu). Prin urmare elementele unei astfel de colecții sunt de fapt perechi de obiecte, dintre care unul serveşte drept cale de acces spre celălalt. Din punctul de vedere al utilizatorului, astfel de colecții sunt „privite" prin interfața *Map*.

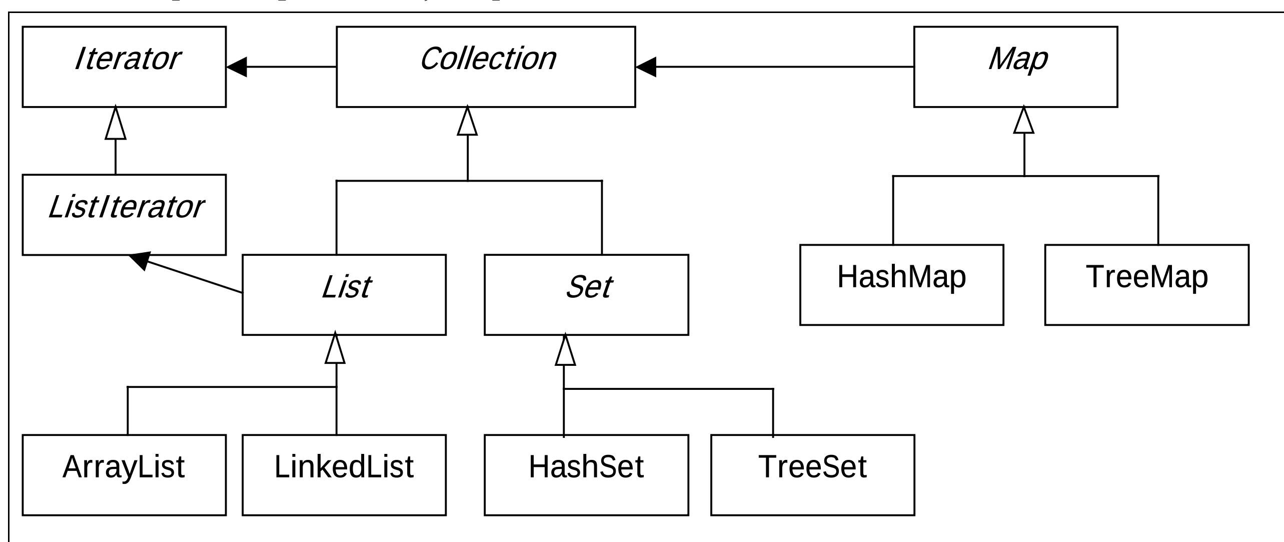


Figura 2-15 O parte din clasele cele mai importante ale API-ului privind colecțiile

Definiția interfeței *Collection* cuprinde, în esență, următoarele:

Collection – interfață	
public int **size**()	Returnează numărul de elemente al colecției

public boolean **isEmpty**()	Verifică dacă există cel puțin un element inițializat în colecție
public boolean **containts**(Object element)	Verifică existența elementului specificat în colecție
public void **add**(Object element)	Adaugă un element în colecție
public void **remove**(Object element)	Şterge un element din colecție
public Iterator **iterator**()	Întoarce o instanță *Iterator* prin ale cărei metode *next()* şi *hasNext()* se poate constitui o buclă iterativă pentru parcurgerea colecției
public boolean **addAll**(Collection c)	Adaugă în colecția curentă elementele altei colecții
public boolean **removeAll**(Collection c)	Şterge din colecția curentă toate elementele care se găsesc şi în colecția specificată (prin argument)
public boolean **retainAll**(Collection c)	Şterge din colecția curentă toate elementele, cu excepția celor care se regăsesc în colecția specificată
public boolean **containsAll**(Collection c)	Verifică dacă toate elementele din colecția specificată se găsesc şi în colecția curentă
public void **clear**()	Goleşte colecția
public Object[] **toArray**()	Creează un *array* pe baza elementelor colecției curente.
public boolean **equals**(Object o)	
public int **hashCode**()	Returnează valoarea codului *hash* pentru această colecție (vezi *HasMap)*

Prin urmare, indiferent de clasa care reprezintă materializarea fizică a colecției, utilizatorii aceştia o pot acces în orice situație folosind metodele din definiția interfeței de mai sus.

Container-ele de tip *Colection* sunt diferențiate la rândul lor în două categorii:

- colecții ordonate de elemente *neduplicate*, caz în care accesarea acestora se face prin intermediul interfeței *Set* şi
- colecții ne-ordonate de elemente păstrate în ordinea adăugării, interfața *List* fiind esențială în acest caz.

Definiția interfeței **List** (bineînțeles *List* extinde interfața *Collection,* după cum rezultă şi din figura 6.2) este următoarea:

List – interfață	
public boolean **addAll**(int, colection)	Inserează toate elementele din colecția specificată (prin argument) în colecția inițială începând la o anumită poziție
public Object **get**(int)	Returnează elementul de la poziția (indexul)

	specificat
public Object **set**(int, Object)	Înlocuieşte elementul de la poziția indicată cu obiectul specificat (prin al doilea argument). Returnează elementul care se găsea anterior pe respectiva poziție
public Object **remove**(int)	Îndepărtează din colecție elementul de la poziția specificată. Returnează elementul eliminat.
public int **indexOf**(Object)	Returnează poziția la care apare prima dată în colecție elementul specificat sau –1 în cazul insuccesului.
public int **lastIndexOf**(Object)	Returnează poziția la care apare ultima dată în colecție elementul specificat sau –1 în cazul insuccesului.
public ListIterator **listIterator**()	Returnează o instanță ListIterator (subclasă a Iterator) pentru parcurgerea elementelor din listă.
public ListIterator **listIterator**(int)	Returnează o instanță ListIterator (subclasă a Iterator) pentru parcurgerea elementelor din listă începând cu poziția specificată.
public List **subList**(int *from*, int *to*)	Returnează o nouă colecție de tip List ale cărei elemente sunt preluate din colecția inițială între pozițiile specificate

Față de interfața *Collection*, *List* adaugă în special metode destinate accesului la elementele individuale ținând seama de faptul că indexul fiecăruia este asociat consecutiv funcție de adăugarea în listă. Parcurgerea elementelor se poate face clasic, folosind indexul, sau poate fi realizată într-o altă ordine dictată de o instanță *ListIterator*.

Un *Iterator* reprezintă o intefață pentru care clasele concrete trebuie să implementeze metodele:

```
public boolean hasNext();
public Object next();
public void remove();
```

Un *ListInterator* extinde interfața Iterator şi este asociat exclusiv *List*-elor. El adăugă metode opționale prin care poate gestiona elementele listei (add(), set(), remove()), precum şi modalități de parcurgere bidirecțională a listei (hasPrevious(), previous(), nextIndex(), previousIndex()).

Biblioteca privind colecțiile (*java.util*) oferă două clase concrete pentru implementarea listelor:

- *LinkedList* – accesul secvențial este bine optimizat, iar operațiile de inserare şi ştergere din interiorul listei sunt eficiente. Accesul aleator la elementele listei este însă relativ ineficient din punctul de vedere al timpului.

- ***ArrayList*** – care se bazează pe un *Array* şi permite un acces aleator rapid la oricare element colecţie. Inserarea şi eliminarea elementelor din „mijlocul" listei este însă costisitoare.

Un *Set* este o colecţie (*Collection*) care nu prezintă elemente duplicate. Ca urmare, interfaţa *Set* nu prezintă elemente deosebite faţă de interfaţa *Collection* cu excepţia caracteristicii amintită anterior. Pprin urmare dubla adăugare (*add(Object)*) a unui element într-un set nu va avea ca efect duplicarea acestuia, a doua operaţie *add()* neavând nici un efect.

Distribuţia Java oferă două căi de obţinere a unui *Set* folosind următoarele clase concrete:

- *HashSet* – pentru seturile în care timpul de căutare este esenţial. Obiectele care formează elementele unui astfel de set trebuie să definească corespunzător metoda *hashCode()*.
- *TreeSet* – reprezintă un set ordonat bazat pe o structură de tip arbore, aşa încât elementele pot fi obţinute într-o secvenţă ordonată.

Celălalt tip de container, pe lângă *Collection*, este definit prin interfaţă **Map**. După cum am amintit anterior, o astfel de structură de date se bazează pe configurarea elementelor ca perechi *cheie-valoare*. Fiecare cheie are asociată cel mult o singură valoare. Definiţia acestei interfeţe este următoarea:

Map – interfaţă	
public Set **keySet**()	Returnează cheile ca un set
public Collection **values**()	Returnează valorile ca o colecţie (Collection)
public Set **entrySet**()	Returnează elementele cheie-valoare (instanţele *java.util.Map.Entry*) ca un Set obişnuit
public boolean **containsKey**(Object)	Verifică existenţa unei chei
public boolean **containsValue**(Object)	Verifică existenţa unei valori
public Object **get**(Object key)	Returnează o valoare cunoscându-i cheia asociată
public Object **put**(Object key, Object value)	Asociază valoarea specificată cu cheia precizată prin primul argument. Returnează valoarea asociată anterior cu respectiva cheie.
public void **putAll**(Map)	Copiază toate mapările din containerul *Map* specificat în cel curent
public Object **remove**(Object key)	Elimină valoarea asociată cu cheia specificată prin argument. Returnează elementul (valoarea) eliminată.
public boolean **isEmpty**()	Verifică dacă există asocieri cheie-valoare
public int **size**()	Numărul de asocieri chei-valoare existente
public void clear()	
public boolean equals()	
public int hashCode()	

Un astfel de container reprezintă de fapt o colecție de asociații cheie-valoare. Fiecare element al acestui tip de colecție, sau fiecare pereche cheie-valoare, sau fiecare intrare, reprezintă un obiect al cărui tip este definit prin interfața *java.util.Map.**Entry***:

```
public Object getKey();
public Object getValue();
public Object setValue(Object);
public boolean equals(Object);
public int hashCode();
```

După cum se observă din tabelul de mai sus, un container de tip *Map* poate fi văzut din trei perspective, toate trei fiind de tip *Collection*: ca un *Set* de chei, ca o *Collection* de valori, ca un *Set* de asociații cheie-valoare. Prin urmare parcurgerea unui *Map* se va realiza funcție de *iteratorul* colecție sub care este văzut, printr-o structură de genul:

```
Set s = map.entrySet();
Iterator i = s.Iterator();
while(i.hasNext(){ i.next; ...}
```

Distribuția Java oferă pentru acest tip de container două implementări concrete prin clasele:

- ***HashMap*** – se bazează pe o tabelă de tip *hash* şi furnizează o performanță uniformă pentru inserarea şi localizarea perechilor de valori;
- *TreeMap* – se bazează pe o structură arborescentă, astfel încât cele trei perspective ale containerului vor putea fi obținute deja ordonate.

2.6.1.2 *Clasa* ArrayList

Clasa *ArrayList* implementează, după cum am văzut anterior, interfața *List* la care va adăuga bineînțeles constructorii necesari obținerii concrete a unei astfel de colecții. Definiția acestei clase este următoarea:

```
public ArrayList();
public ArrayList(int size);
public ArrayList(Collection c);
public void trimToSize();
public void ensureCapacity(int minSize);
public Object clone();
```

Prin urmare o astfel de listă poate fi obținută specificându-i, sau nu, dimensiunea inițială (numărul inițial de poziții rezervate), spre deosebire de cazul *array*-urilor unde era absolut necesară cunoaşterea numărului de elemente al colecției. De asemenea, un *ArrayList* se poate obține şi pe baza unei colecții deja existente. Dacă dimensiunea inițial rezervată a fost prea mare, existând un număr de „rubrici” goale, elementele nefolosite pot fi eliminate prin *trimToSize()*, iar dacă inițial nu a fost specificată capacitatea listei, sau cea inițială urmează a fi depăşită, se poate asigura rezervarea unui număr de elemente prin metoda *ensureCapacity()*.

Iată un exemplu de declarare, populare şi acces al unei colecţii *ArrayList*:

```
public class TestArrayList {
   public static void main(String[] args) {
      List list = new ArrayList(2);
      list.add("primul");
      list.add("al doilea");
      list.add("al treilea"); // am depasit deja capacitatea initiala, asa ca
      // lista va final obligata sa-si redefineasca automat dimensiunea
      list.add(3, "al patrulea"); // operatie validata fiindca urmeaza indexul 3
      //list.add(5, "primul"); // operatie validata fiindca urmeaza indexul 4
      list.add(4, "al cincilea");
      list.add(2,  "ultimul");

      // Parcurgem lista in stil "clasic":
      System.out.println("Iteratie clasica folosind for()");
      for(int i=0; i<list.size(); i++)
         System.out.println(list.get(i));

      // Parcurgem lista prin interator in ordinea indexarii:
      System.out.println("Iteratie cu iterator in ordinea indexarii");
      // Obtin un iterator care porneste de la primul index
      Iterator iterator = list.listIterator();
      while(iterator.hasNext())
         System.out.println(iterator.next());

      // Parcurgem lista prin interator in ordine inversa:
      System.out.println("Iteratie cu iterator in ordinea inversa indexarii");
      // Reconsider iteratorul anterior ca ListIterator, si care
      // in iteratia de mai jos va porni de la ultimul element, la
      // care a ajuns prin iteratia de msi sus
      ListIterator listIterator = (ListIterator)iterator;
      while(listIterator.hasPrevious())
         System.out.println(listIterator.previous());
   }
}
```

Rezultatul va fi următorul:

```
Iteratie clasica folosind for()
primul
al doilea
ultimul
al treilea
al patrulea
al cincilea
Iteratie cu iterator in ordinea indexarii
primul
al doilea
ultimul
al treilea
al patrulea
al cincilea
Iteratie cu iterator in ordinea inversa indexarii
al cincilea
al patrulea
al treilea
ultimul
al doilea
primul
```

2.6.1.3 *Clasa* HashMap

Clasa *HashMap* implementează, după cum am arătat anterior, interfaţa *Map* oferind avantajul asocierii elementelor cu orice fel de tip de valori (obiecte), nelimitându-se la valorile primitive întregi (int) specifice indecşilor celorlalte tipuri de colecţii (*Array* şi *Collection*). De asemenea, accesul valorilor folosind perechile-chei se face la performanţe rezonabilă, ceea ce face din acest tip de container unul dintre cele mai folositoare.

În definiţia clasei *HashMap* se găsesc, pe lângă metodele implementate din interfaţa *Map*, în primul rând metodele constructor pentru obţinerea concretă a unei astfel de structuri:

```
public HashMap();
public HashMap(int size, float load);
public HashMap(int size);
public HashMap(Map);
```

Parametrul *size* din constructorii de mai sus semnifică capacitatea iniţială a tabelei, iar parametrul *float* specifică dimensiunea la care trebuie să ajungă tabela pentru a fi relocalizată în altă zonă de memorie.

Iată un exemplu concret de folosire a acestei clase:

```
public class TestHashMap {
    public static void main(String[] args) {
        String[] coduri = {"IS", "BC", "PN", "BT", "SV"};
        String[] judete = {"Iasi", "Bacau", "Piatra-Neamt", "Botosani", "Suceava"};
        Map indicative = new HashMap();

        // populez indicative
        for (int i=0; i<coduri.length; i++)
            indicative.put(coduri[i], judete[i]);

       // parcurg cele trei perspective ale containerului
       // mai intai cheile
       Set chei = indicative.keySet();
       Iterator iterator = chei.iterator();
       System.out.println("Cheile:");
       while(iterator.hasNext())
           System.out.println(iterator.next());
       // apoi valorile
       Collection valori = indicative.values();
       iterator = valori.iterator();
       System.out.println("Valorile:");
       while(iterator.hasNext())
           System.out.println(iterator.next());
       // in fine setul de perechi chei-valoare
       Set intrari = indicative.entrySet();
       iterator = intrari.iterator();
       System.out.println("Perechile cheie-valoare:");
       while(iterator.hasNext()){
           java.util.Map.Entry intrare = (java.util.Map.Entry) iterator.next();
           System.out.println(intrare.getKey() + "-" + intrare.getValue());
       }
       // extrag o valoare cunoscindu-i cheia
       String id = "IS";
       System.out.println("Indicativul " + id + " corespunde judetului " + indicative.get(id));
    }
}
```

Iar rezultatul va fi următorul:

```
Cheile:
BT
```

```
PN
BC
IS
SV
Valorile:
Botosani
Piatra-Neamt
Bacau
Iasi
Suceava
Perechile cheie-valoare:
BT-Botosani
PN-Piatra-Neamt
BC-Bacau
IS-Iasi
SV-Suceava
Indicativul IS corespunde judetului Iasi
```

În exemplul anterior, în condițiile în care cheia este de tip *String*, lucrurile au decurs fără probleme şi rezultatul este cel scontat. Să modificăm însă clasele implicate în perechile cheie-valoare de mai înainte şi să verificăm din nou selecția unui element cunoscându-i cheia. Presupunem că structura claselor este următoarea:

```
class Cod{
   private int nrCod;
   private String sirCod;
   public Cod(int pNrCod, String pSirCod){
      nrCod = pNrCod;
      sirCod = pSirCod;
   }
   public String toString(){
      return sirCod;
   }
   public Integer indicativNumeric(){
      return new Integer(nrCod);
   }
}
class Judet{
   private String nume;
   private String resedinta;
   public Judet(String pNume, String pResedinta){
      nume = pNume;
      resedinta = pResedinta;
   }
   public String toString(){
      return nume;
   }
   public String getResedinta(){
      return resedinta;
   }
}
```

Dacă realizăm următorul test:

```
public static void main(String[] args) {
   Cod[] coduri = {new Cod(232, "IS"), new Cod(234,"BC"), new Cod(233, "PN")
      , new Cod(231, "BT"), new Cod(230, "SV")};
   Judet[] judete = {new Judet("Iasi", "Iasi"), new Judet("Bacau", "Bacau"),
      new Judet("Piatra", "Piatra-Neamt"), new Judet("Botosani", "Botosani"),
      new Judet("Suceava", "Suceava")};
   Map indicative = new HashMap();
```

```
        // populez indicative
        for (int i=0; i<coduri.length; i++)
            indicative.put(coduri[i], judete[i]);

        // extrag o valoare cunoscindu-i cheia
        Cod cod = new Cod(232, "IS");
        if (indicative.containsKey(cod))
            System.out.println("Indicativul " + cod +
            " corespunde judetului " + indicative.get(cod));
        else
            System.out.println("Nu gasesc cheia " + cod);
    }
```

rezultatul va fi următorul:

```
Nu gasesc cheia IS
```

Deşi la prima vedere rezultatul ne poate face neîncrezători, totuşi există o explicaţie destul de plauzibilă. Aşa cum îi spune de altfel şi numele, un *HasMap* foloseşte pentru regăsirea elementelor pereche din colecţie valoarea *hash* asociată cheii, adică, mai exact, rezultatul funcţiei *hashCode()* moştenită de la clasa Object. Implicit, această valoare se bazează pe adresa de memorie a obiectului respectiv. Prin urmare cele două instanţe obţinute anterior prin aceeaşi instrucţiune *new Cod(232, „IS”)* reprezentă obiecte diferite, deci şi codurile lor *hash* sunt altele. Deci va trebui să rescriem metoda *hashCode()* în clasa *Cod* astfel încât în ambele cazuri să returneze aceeaşi valoare:

```
    public int hashCode(){
        return indicativNumeric().intValue();
    }
```

Şi totuşi chiar şi după această manevră nu avem încă succes în cazul testului anterior. Acest lucru se datorează faptului că valoarea cheii după care am accesat colecţia nu este considerată egală cu valoarea cheii regăsite prin intermediul *hashcode*-ului. Pentru a rezolva şi acest incovenient va trebui să rescriem şi metoda *equals()* a clasei *Cod* astfel:

```
    public boolean equals(Object o){
        return (o instanceof Cod)&&
            (this.indicativNumeric().intValue() == ((Cod)o).indicativNumeric().intValue());
    }
```

În final, dacă vom re-testa căutarea în *HashMap*-ul construit pe baza *Cod*-urilor şi *Judeţ*-elor vom obţine un rezultat rezonabil:

```
Indicativul IS corespunde judetului Iasi
```

ca urmare a faptului că în acest moment configuraţia clasei *Cod,* care reprezintă cheia de căutare, este următoarea:

```
class Cod{
    private int nrCod;
    private String sirCod;
    public Cod(int pNrCod, String pSirCod){
        nrCod = pNrCod;
        sirCod = pSirCod;
    }
    public String toString(){
        return sirCod;
    }
    public Integer indicativNumeric(){
        return new Integer(nrCod);
```

```
    }

    public int hashCode(){
       return indicativNumeric().intValue();
    }

    public boolean equals(Object o){
       return (o instanceof Cod)&&
          (this.indicativNumeric().intValue() == ((Cod)o).indicativNumeric().intValue());
    }
}
```

2.7 Persistenței obiectelor: serializarea

De regulă, pentru aplicațiile economice, persistența este necesară pentru entitățile care formează „domeniul" problemei, caz în care soluția la care se recurge este o bază de date (de cele mai multe ori relațională) legată de „spațiul", să-i zicem tranzient, al aplicației prin intermediul claselor din biblioteca JDBC. În acest sens au proliferat o serie întreagă de „framework"-uri pentru asigurarea transparenței salvării/reconstituirii obiectelor din spațiul bazelor de date, două din cele mai cunoscute fiind JDO (Java Data Objects) şi EJB (Enterprise Java Beans cu cele două categorii de componente BMP şi CMP – Bean Managed Persistence şi Container Managed Persistence). Problema persistenței este însă o problemă mai generală şi poate fi asigurată şi prin mijloace mai puțin sofisticate, de exemplu *serializând* obiectele în fişiere ale sistemului de operare.

2.7.1 Suportul privind gestiunea fişierelor şi scrierii datelor în fişiere

Lucrul cu fişiere în Java este asigurat prin API-ul *java.io* care asigură clasele necesare abstractizării atât fişierelor sistemului de operare (localizare, deschidere, creare, ştergere fişiere sau directoare) cât şi a fluxului de date, adică citirea conținutului fişierelor şi salvarea datelor în fişiere.

Filozofia de proiectare a fluxurilor I/O în Java are în vedere următoarele principii:

- operațiile de I/O (intrare-ieşire) se bazează pe fluxuri sau canale de date numite *stream*-uri. Un astfel de „canal" abstractizează un dispozitiv specific poziționat la unul din capete şi se specializează funcție de tipul şi calitatea datelor transportate prin intermediul unor clase specifice;
- există câteva aspecte care diferențiază platformele (sistemele de operare), cum ar fi delimitatorii căilor de directoare, de aceea programele trebuie scrise într-o manieră care să evite astfel de incompatibilități la rularea pe sisteme diferite;
- clasele care specializează fluxurile de date (unele pentru citirea datelor, altele care lucrează cu un anume tip de date, altele care controlează citirea din fişiere etc.) sunt construite de o manieră telescopică astfel încât construirea unora se realizează prin compunere sau combinare cu altele pe care le specializează.

Într-o imagine succintă, biblioteca *java.io* arată cam în felul următor:

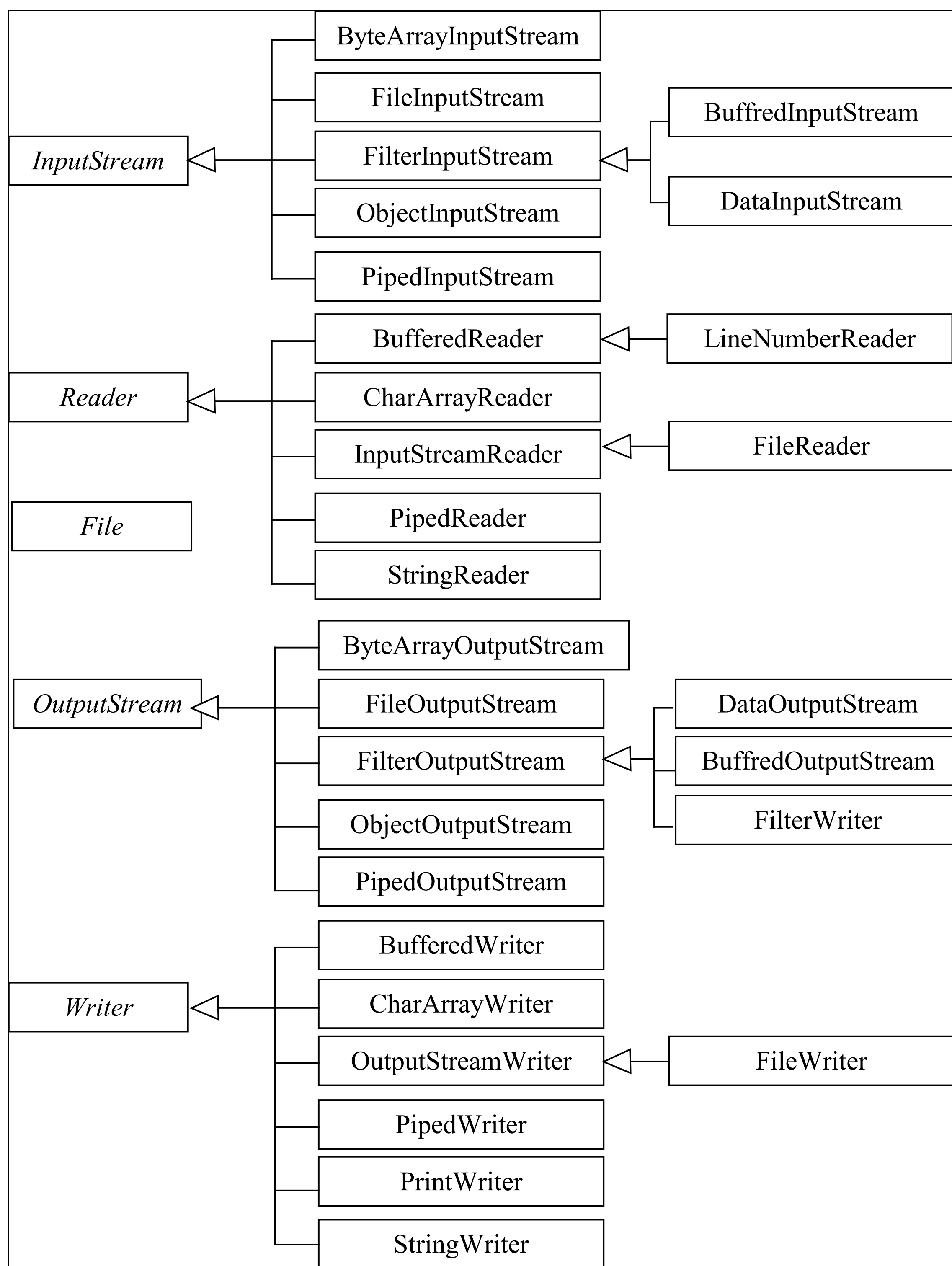


Figura 2-16 Biblioteca *java.io*

După cum se observă din figura anterioară, biblioteca *java.io* are o anumită complexitate, însă în continuare vom discuta doar caracteristicile generale ale acesteia şi vom particulariza câteva din clasele mai importante.

De la bun început trebuie să remarcăm că există două categorii de clase destinate, pe de o parte, ***citirii*** (*InputStream* şi *Reader*) şi, pe de altă parte, ***scrierii*** (*OutputStream* şi *Writer*). O a doua diferenţă esenţială se mai face între subclasele care manipulează pentru citire sau scriere date binare (*InputStream* şi *OutputStream*) şi cele care manipulează text (*Reader* şi *Writer*).

De asemenea, atunci când datele binare care constituie obiectul prelucrărilor sunt de tip *String* sau de un tip primitiv (char, boolean, int double, float), atunci cel mai des se apelează la subclasele *DataInputStream* şi *DataOutputStream*, echivalente cu *Reader* şi *Writer* în privinţa datelor tip text.

Atunci când datele provin sau vor fi trimise în fişiere, sunt folosite clasele specializate *FileInputStream* şi *FileOutputStream*, iar atunci când diversele date de tip text sau de tipuri primitive (int, float, double etc.) sunt direcţionate către o consolă de ieşire, este folosită clasa *PrintWriter* (de exemplu *Sytem.out* de tip *PrintWriter*).

Filtrarea datelor de intrare sau de ieşire se referă la a efectua diverse operaţii asupra lor înainte de a fi introduse în fluxul de intrare sau ieşire, în acest sens de folos fiind clasele *FilterInputStream*, *FilterOutputStream* şi *FilterWriter*.

O altă operaţie specifică (de filtrare) se referă la preluarea sau trimiterea datelor în ***blocuri – date bufferizate***. În acest sens întâlnim clasele *BufferedInputStream* şi *BufferedReader* pentru citirea datelor binare sau text, şi *BufferedOutputStream* şi *BufferedWriter* pentru scrierea lor.

În medii cu mai multe fire de execuţie (threads) active simultan (multi-threading) transferul de date între acestea se efectuează prin intermediul canalelor de tip *pipe*. În acest scop vor fi folosite clasele *PipedInputStream, PipedReader, PipedOutputStream* şi *PipedWriter*.

În fine, mai există o serie de clase care furnizează operaţii de tip I/O nu pentru fişiere, console sau fire de execuţie, ci pentru structuri de memorie de tip *Array*. Clasele dedicate pentru astfel de operaţii sunt *ByteArrayInputStream*, *ByteArrayOutputStream* pentru date binare şi *CharArrayReader, CharArrayWriter* pentru text.

2.7.1.1 Lucru cu fişiere

Dacă avem în vedere clasele pentru constituirea fluxurilor de citire sau scriere în fişiere va trebui mai întâi să ne oprim asupra clasei *File* care abstractizează la nivelul aplicaţiilor Java fişierele (sau directoarele) platformei de operare. Această clasă furnizează următoarea funcţionalitate:

- oferă acces la informaţiile prin care se pot determina: numele, cale din ierarhia de directoare, dimensiunea, modul de acces permis (citire, citire/scriere), tipul (fişier sau director) ale obiectului din sistemul de fişiere vizat. În acest sens există metodele: *getName(), getPath(), getParent(), getParentFile(), getAbsolutePath(), canRead(), canWrite(), isDirectory(), isFile(), length()* etc.
- permite efectuarea operaţiilor de creare a unor fişiere/directoare noi, ştergere a unor fişiere/directoare existente, prin metodele *createNewFile(), delete(), renameTo()* etc.

Prin urmare, pentru a crea un nou fişier la nivelul sistemului de operare se poate proceda în felul următor:

```
import java.io.*;
import javax.swing.JFileChooser;
/**
 *
 * @author  strimbeic
 */
public class TestFisiere {
```

```
    private static boolean isWritableFile(String fileName){
        try{
            File fisier = new File(fileName);
            if (!fisier.exists())
                throw (new FileNotFoundException("Nu gasesc fisierul cu numele " + fileName));
            if (!fisier.canWrite())
                throw (new IOException("Fisier " + fileName + " neaccesibil la citire"));
            return true;
        }
        catch(FileNotFoundException e){
            System.out.println("ERORARE fisier inexistent: " + e.getMessage());
            return false;
        }
        catch(IOException e){
            System.out.println("ERORARE de acces: " + e.getMessage());
            return false;
        }
    }
    private static void createNewFile(String fileName){
        try{
            File fisier = new File(fileName);
            System.out.println("Creez fisier " + fisier.getPath());
            fisier.createNewFile();
        }catch(IOException e){
            System.out.println("Esec creare fisier !");
        }
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        JFileChooser dialog = new JFileChooser();
        int result = dialog.showDialog(null, "New");
        File fisierSelectat = dialog.getSelectedFile();

        String numeCompletFisier = fisierSelectat.getParent() + "\\" + fisierSelectat.getName();
        createNewFile(numeCompletFisier);
        if (isWritableFile(numeCompletFisier))
            System.out.println("OK");
    }
}
```

În legătură cu listingul de mai sus, se remarcă faptul că pentru selectarea directorului în care să fie creat noul fişier am folosit clasa *JFileChooser*. Aceasta poate fi folosită pentru a afişa un dialog din care se poate selecta o cale sau un fişier din sistemul de operare, oferind în acest sens trei metode *showDialog(componentă, numeButon)* – care oferă posibilitatea denumirii butonului principal de „validare" folosind o etichetă personalizată, *showOpenDialog()* şi *showSaveDialog()*. Prin urmare, la execuție va fi indicată calea unde va fi creat noul fişier şi numele acestuia în caseta *File Name* a dialogului. Numele complet (calea directorului părinte şi numele simplu) al noului fişier este trimisă ca parametru metodei *createNewFile()* unde este creat un obiect de tip *File* a cărui metodă *createNewFile()* va avea responsabilitatea finală. În fine prin metoda *isWritableFile()* este verificată existența noului fişier şi disponibilitatea lui la citire.

2.7.1.2 Scriere şi citire date binare

În momentul în care se decide **scrierea datelor** în fişiere, trebuie clarificate câteva detalii esenţiale în alegerea claselor care au capacitatea să realizeze corect această operaţie:

- tipului operaţiei - evident, de scriere;
- tipul de date care vor rezulta în urma operaţiei – să presupunem că este vorba de date binare;
- tipului recipientului (sau furnizorului de date), în cazul de faţă – fişier.

Prin urmare, clasele necesare unei operaţii de scriere de date binare într-un fişier sunt:

- o subclasă *OutputStream* (ieşire) pentru date binare – adică *DataOutputStream;*
- o clasă care să direcţioneze fluxul de date spre un fişier: *FileOutputStream*.

Definiţia clasei *DataOutputStream* este, pe scurt, următoarea:

DataOutputStream	
public DataOutputStream(OutputStream s)	Constructorul clasei care permite combinarea cu un alt *OutputStream*, de exemplu cu *FileOutputStream*
public flush()	Metoda care trimite datele spre ieşire în cazul fluxurilor buferizate
public writeBoolean(Boolean b)	Metode pentru scriere a fiecărui tip de dată primitiv din Java
public writeByte(int i)	
public writeBytes(String s)	
public writeChar(int c)	
public writeChars(String s)	
public writeDouble(double d)	
public writeFloat(float f)	
public writeInt(int i)	
public writeLong(long l)	
public writeShort(short s)	
public writeUTF(String s)	Metoda pentru scrierea şirurilor de caractere (String) în format Unicode

Clasa *FileOutputStream* are ca responsabilitate esenţială instanţierea de fluxuri de ieşire către un fişier de date al cărui nume sau referinţă este specificat (specificată) în constructor. Metoda de scriere – *write*() – se referă doar pentru tipuri de date *int* şi *byte*, motiv pentru care, în cazul şirurilor de caractere de exemplu, este necesară folosirea unei clase suplimentare *DataOutpuStream*.

Obţinerea unui flux de scriere date binare într-un fişier s-ar putea realiza astfel:

```
DataOutputStream outStream = new DataOutputStream(new FileOutputStream(fişier));
```

Prin urmare am putea adăuga clasei *TestFisiere* o metodă de scriere care ar putea să arate astfel:

```
private static void salveazaDateBinar(String date, File fisierSalvare){
   try{
      DataOutputStream outStream =
         new DataOutputStream(new FileOutputStream(fisierSalvare));
      outStream.writeUTF(date);
      //outStream.flush();
      outStream.close();
   }catch(Exception e){
      System.out.println("Eroare de scriere: " + e.getMessage());
   }
}
```

Citirea datelor este o operaţie inversă scrierii şi, ca urmare ar trebui să există o serie de clase pereche faţă de cele anterioare, şi anume *DataInputStream* şi *FileInputStream*. Definiţia clasei *DataInputStream* este următoarea:

DataInputStream	
public DataInputStream(InputStream s)	Constructorul clasei care permite combinarea cu un alt *InputStream*, de exemplu cu *FileInputStream*
public boolean readBoolean()	Metode pentru citire a fiecărui tip de dată primitiv din Java
public byte readByte()	
public char readChar()	
public double readDouble()	
public float readFloat()	
public int readInt()	
public long readLong()	
public short writeShort()	
public String readUTF()	Metoda pentru citirea şirurilor de caractere (String) în format Unicode

Clasa prin care se va deschide pentru citire un fişier de date binare este *FileOutputStream*.

În definiţia clasei *DataInputStream* se observă faptul că metodele de citire sunt pereche faţă de metodele de scriere din *DataOutputStream*. Citirea dintr-un astfel de fişier pune problema determinării momentului încetării operaţiei de citire, sau a sfârşitului fişierului. Spre deosebire de fişierele de text, fişierele de date binare nu conţin marcatorul EOF (End-Of-File) şi ca urmare în algoritmul de citire se va ţine seama de o eroare anticipată, şi anume *EOFException*:

```
private static String citesteDateBinar(File fisierSursa){
   String sirCitit = null;
   try{
      DataInputStream inStream =
         new DataInputStream(new FileInputStream(fisierSursa));
      sirCitit = inStream.readUTF();
      inStream.close();
   }
   catch(EOFException e){}
   catch(Exception e){System.out.println("Eroare de citire " + e.getMessage());}
   return sirCitit;
}
```

2.7.1.3 Scriere şi citire text

După cum am arătat mai înainte, pentru scrierea şi citirea datelor de tip text vom folosi clasele *Writer* şi *Reader* sau subclasele lor.

Pentru a obține un flux de date dintr-un fişier atunci vom folosi clasa *FileWriter* derivată din clasa *Writer* prin intermediul subclasei *OutputStreamWriter*. Clasa (părinte) *Writer* defineşte o metodă *write(String s)* care va fi folosită pentru a trimite date în fluxul de scriere.

Ca urmare o metodă de scriere, de data aceasta a datelor de tip text, ar putea arăta astfel:

```
private static void salveazaText(String date, File fisierSalvare){
   try{
      FileWriter outStream =
         new FileWriter(fisierSalvare);
      outStream.write(date);
      outStream.close();
   }catch(Exception e){
      System.out.println("Eroare de scriere: " + e.getMessage());
   }
}
```

Pentru citirea datelor dintr-un fişier text vom avea nevoie de clasa *FileReader*, subclasă a clasei *Reader*, a cărei principală responsabilitate este deschiderea unui fişier text pentru citire. Clasa *Reader* defineşte o metodă *read()* pentru reconstituirea datelor din sursa de date. Problema acestei metode este că citeşte un singur caracter (întoarce un *int* care poate fi convertit prin casting într-un char) şi nu o linie întreagă. Singura clasă care conține o metodă *readLine()* ce va citi o întreagă linie, dintr-un flux de ieşire, ca un şir de caractere este *BufferedReader*, motiv pentru care va trebui să combinăm această clasă cu *FileReader*. Ca urmare, pentru a obține un flux de date dintr-un fişier din care să citim linie cu linie se poate proceda astfel:

```
BufferedReader inStream = new BufferedReader(new FileReader(fisier));
```

Dacă fluxul de intrare conține mai multe linii, metoda de citire ar putea arăta astfel:

```
public static String citesteText(File fisierSursa){
   StringBuffer textCitit = new StringBuffer();
   try{
      BufferedReader inStream =
         new BufferedReader(new FileReader(fisierSursa));
      String linie = inStream.readLine();
      while (linie!=null){
         textCitit.append(linie);
         linie = inStream.readLine();
      }
      inStream.close();
   }
   catch(EOFException e){}
   catch(Exception e){System.out.println("Eroare de citire " + e.getMessage());}

   return textCitit.toString();
}
```

Metoda *main()* din clasa *TestFisiere*, folosită pentru a putea testare, ar putea fi redefinită astfel:

```
public static void main(String[] args) {
   JFileChooser dialog = new JFileChooser();
```

```
        // pentru fisiere binare
        int result = dialog.showDialog(null, "New fisier binar");
        File fisierSelectat = dialog.getSelectedFile();
        String numeCompletFisier = fisierSelectat.getParent() + "\\" + fisierSelectat.getName();
        createNewFile(numeCompletFisier);
        if (isWritableFile(numeCompletFisier))
           System.out.println("OK creare fisier");
        System.out.println("Scrie in fisier binar: date salvate");
        salveazaDateBinar("date salvate", fisierSelectat);
        System.out.println("Citesc din fisier binar:" + citesteDateBinar(fisierSelectat));

        // pentru fisier text
        result = dialog.showDialog(null, "New fisier text");
        fisierSelectat = dialog.getSelectedFile();
        numeCompletFisier = fisierSelectat.getParent() + "\\" + fisierSelectat.getName();
        createNewFile(numeCompletFisier);
        if (isWritableFile(numeCompletFisier))
           System.out.println("OK creare fisier");
        System.out.println("Scrie in fisier text: date salvate");
        salveazaText("date salvate", fisierSelectat);
        System.out.println("Citesc din fisier binar:" + citesteText(fisierSelectat));
    }
```

2.7.2 Serializarea obiectelor

Exemplele anterioare au arătat cum pot fi salvate şi reconstituite date din fişiere binare sau conţinând text. Limbajul Java prezintă însă şi o modalitate, relativ transparentă, de a salva şi reconstitui obiecte în/din fişiere. În acest scop vor fi folosite tot două subclase *OutputStream* respectiv *InputStream*, la fel ca în cazul datelor binare, însă acestea vor fi *ObjectOutputStream* şi *ObjectInputStream* necesare în procesul de ***serializare*** a obiectelor. Pentru ca un obiect să fie recunoscut ca *serializabil* el va trebui să implementeze interfaţa Serializable, fapt ce reprezintă doar marcarea acelei clase ca având obiecte ce pot avea caracteristici de persistenţă. În acest scop vom construi metodele *writeToFile(FileOutputStream s)* şi *readFromFile(FileInputStream s)*. Logica de salvare/reconsituire a obiectelor prin acestea se bazează cel mai adesea pe clasele *ObjectOutputStream* şi *ObjectInputStream*, dar ar putea fi proiectat şi un mod de folosire mai directă a fişierelor, după cum a fost prezentat în paragrafele anterioare.

Clasa *ObjectOutputStream* instanţiază fluxuri de date preluând în constructor un *FileOutputStream* şi prezintă o metodă de scriere *writeObject(Object o)* care va avea responsabilitatea convertirii în date binare a valorilor atributelor (non-*transient*-e) şi salvarea lor pe disc.

Clasa *ObjectInputStream* instanţiază fluxuri de date preluând în constructor un *FileInputStream* şi prezintă o metodă de citire *readObject()* care va avea responsabilitatea reconstituirii obiectelor prin convertirea datelor binare în tipurile corespunzătoare valorilor atributelor (non-*transient*-e).

O clasă serializabilă care ar putea să beneficieze de serviciile unor clase cum sunt *ObjectOutputStream* şi *ObjectInputStream* ar putea avea configuraţia următoare:

```
class Student implements Serializable{
   public String matricol;
   public String nume;
   public String prenume;
   public Student(){}
```

```
    public Student(String pMatricol, String pNume, String pPrenume){
        matricol = pMatricol;
        nume = pNume;
        prenume = pPrenume;
    }

    public void writeToFile(FileOutputStream outStream) throws IOException{
        ObjectOutputStream oStream = new ObjectOutputStream(outStream);
        oStream.writeObject(this);
        oStream.flush();
    }
    public void readFromFile(FileInputStream inStream) throws IOException,
ClassNotFoundException{
        ObjectInputStream oStream = new ObjectInputStream(inStream);
        Student s = (Student)oStream.readObject();
        this.matricol = s.matricol;
        this.nume = s.nume;
        this.prenume = s.prenume;
    }
  }
```

Pentru a test clasa de mai sus putem proceda astfel:

```
    public static void main(String[] args) {
        // Creez si salvez doua obiecte
        Student s1 = new Student("M001", "Primul", "Student");
        Student s2 = new Student("M002", "Al doilea", "Student");
        JFileChooser dialog = new JFileChooser();
        int result = dialog.showDialog(null, "New fisier");
        File fisier = dialog.getSelectedFile();
        try{
            fisier.createNewFile();
            FileOutputStream outStream = new FileOutputStream(fisier);
            s1.writeToFile(outStream);
            s2.writeToFile(outStream);
            // Reconstitui obiectele
            FileInputStream inStream = new FileInputStream(fisier);
            s1.readFromFile(inStream);
            s2.readFromFile(inStream);
            System.out.println(s1.nume + " " + s1.prenume + " " + s1.matricol);
            System.out.println(s2.nume + " " + s2.prenume + " " + s2.matricol);
        }catch(Exception e){e.printStackTrace();}
    }
```

Totul ar trebui să decurgă fără probleme, iar cei doi studenţi să fie reconstituiţi corect. Totuşi trebuie făcut un mic comentariu: refacerea studenţilor din fişierul de date s-a realizat prin acelaşi flux de ieşire (aceeaşi instanţă *FileInputStream*) şi în aceeaşi ordine în care au fost salvaţi în fişier. Prin urmare, pentru reconstituirea corectă a unui obiect dintr-un fişier, aceasta va trebui căutat după toate celelalte obiecte salvate înaintea lui în fişierul de date. Din acest motiv, ar putea fi adăugată o metodă *getObjects()* care să reconstituie sub forma unei colecţii toate obiectele dintr-un fişier:

```
    public static ArrayList getObjects(FileInputStream inStream) throws ClassNotFoundException{
        ArrayList studenti = new ArrayList();
        try{
            ObjectInputStream oStream;
            Object o;
            while(true){
                oStream = new ObjectInputStream(inStream);
```

```
            o = oStream.readObject();
            studenti.add(o);
        }
    }catch(IOException e){e.printStackTrace();}
    return studenti;
}
```

Pentru testare, metoda *main* prezentată mai sus ar putea fi schimbată astfel:

```
public static void main(String[] args) {
    // Creez si salvez doua obiecte
    Student s1 = new Student("M001", "Primul", "Student");
    Student s2 = new Student("M002", "Al doilea", "Student");
    JFileChooser dialog = new JFileChooser();
    int result = dialog.showDialog(null, "New fisier");
    File fisier = dialog.getSelectedFile();
      try{
        fisier.createNewFile();
        FileOutputStream outStream = new FileOutputStream(fisier);
        s1.writeToFile(outStream);
        s2.writeToFile(outStream);
        // Reconstitui obiectele
        FileInputStream inStream = new FileInputStream(fisier);
        ArrayList studenti = Student.getObjects(inStream);
        Student s;
        for(int i=0; i<studenti.size(); i++){
            s = (Student)studenti.get(i);
            System.out.println(s.nume + " " + s.prenume + " " + s.matricol);
        }
    }catch(Exception e){e.printStackTrace();}

}
```

Nu este obligatorie însă serializarea directă prin mecanismul descris mai sus. Se pot astfel folosi cu succes şi „stream"-uri simple *DataOutputStream* şi *DataInputStream.* În aceste condiții clasa *Student* ar arăta astfel:

```
class Student implements Serializable{
    public String matricol;
    public String nume;
    public String prenume;
    public Student(){}
    public Student(String pMatricol, String pNume, String pPrenume){
        matricol = pMatricol;
        nume = pNume;
        prenume = pPrenume;
    }

    public void writeToFile(FileOutputStream outStream) throws IOException{
        DataOutputStream oStream = new DataOutputStream(outStream);
        oStream.writeUTF(matricol);
        oStream.writeUTF(nume);
        oStream.writeUTF(prenume);
    }
    public void readFromFile(FileInputStream inStream) throws IOException, ClassNotFoundException{
        DataInputStream oStream = new DataInputStream(inStream);
        this.matricol = oStream.readUTF();
        this.nume = oStream.readUTF();
        this.prenume = oStream.readUTF();
    }
    public static ArrayList getObjects(FileInputStream inStream) throws ClassNotFoundException{
        ArrayList studenti = new ArrayList();
        Student o;
```

```
        try{
            DataInputStream oStream = new DataInputStream(inStream);
            while(true){
                o = new Student();
                o.matricol = oStream.readUTF();
                o.nume = oStream.readUTF();
                o.prenume = oStream.readUTF();
                studenti.add(o);
            }
        }
        catch(EOFException e){}
        catch(IOException e){e.printStackTrace();}
        return studenti;
    }
}
```

Jaime Nino, Frederick A. Hosch *An introduction to programming and object oriented design using JAVA*, John Wiley & Sons, Inc. 2002 – ISBN 0-471-35489-9

1. C. Thomas Wu *An introduction to Object-Oriented Programming with java*, WCB/McGraw-Hill 1999, ISBN 0-256-25462-1
2. Cay S. Horstmann, Garry Cornell *Core Java 1.1 Volume I - Fundamentals*, Sun Microsystems Press - Pretince Hall Title - 1997, ISBN 0-13-766957-7
3. Cay S. Horstmann, Garry Cornell *Core Java 1.1 Volume II - Advanced Features*, Sun Microsystems Press - Pretince Hall Title - 1998, ISBN 0-13-766965-8
4. Jeff Schneider & Rajeev Arora *Special Edition Using Enterprise Java*, Que Corporation 1997, ISBN 0-7897-0887-6
5. Bruce Eckel *Thinking in Java*, Second Edition, Pretince Hall PTR, 2000, ISBN 0-13-027363-5
6. Steven Holzner *Java 2 Black Book*, The Coriolis Group, 2001, ISBN 1-58880-097-0
7. Ralph Morelli *Java, Java, Java: Object-Oriented Problem Solving* , Pretince-Hall, Inc, 2003, ISBN 0-13-033370-0